平成13年3月1日発行　DOS/Vmagazine特別編集

SOFTBANK MOOK

2001春号 Oh!X

特集 ポケットの中のコンピュータたち

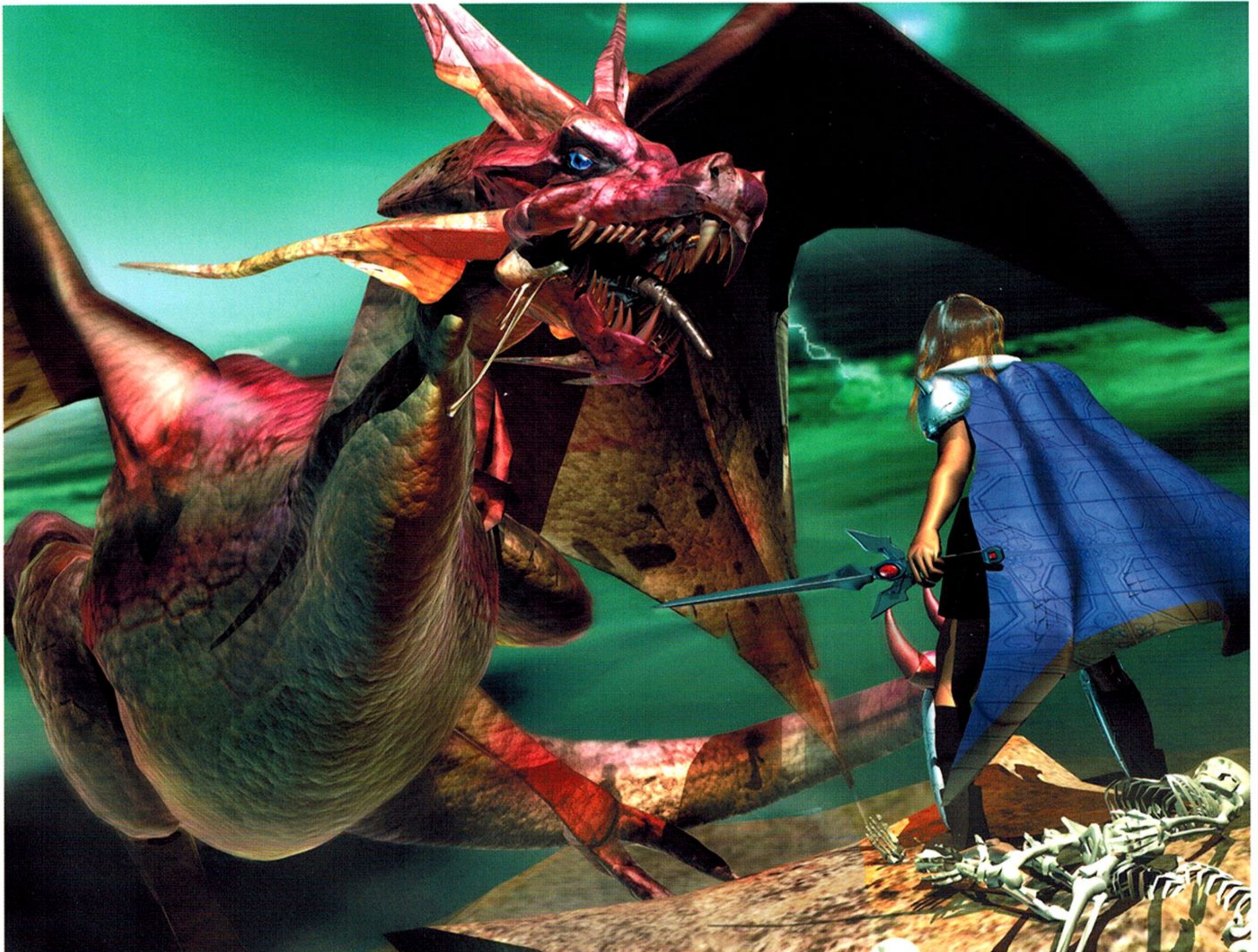
WinCE/Palm/WonderSwanプログラミング/PDAを作る

特別企画 オリジナル筐体第3弾 XThunderbird登場

DirectX8プログラミング入門/Macintoshゲーム制作入門

特別付録 CD-ROM

アニメーションツールDoGA L1/L2/L3

SOFT BANK Publishing
オー！エックス
2001春号
定価2800円

Vwalkerザ・"BIG"モール

Vwalker.com ブイウォーカー・ドット・コム

http://www.vwalker.com/

Web版DOS/V magazine /Vmag.OnLine

ザ・モール The mall

こちらのVwalkerファミリーサイトも
よろしくね！

4GAMER.net for Gamer net

V1ink IT Link Mini Magazine

CLUB AMD AMD User's Community

UNIX USER

PC通

WebΔV

readers' BBS BitWalkers'

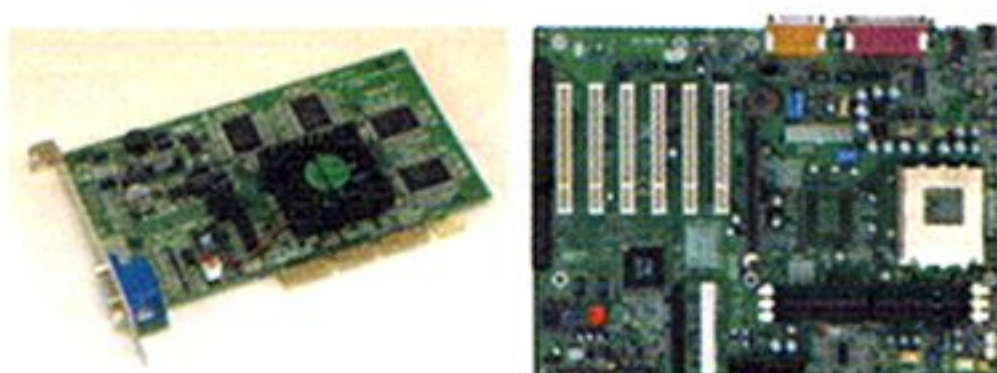
Vwalkerザ・モールは11月29日より，PCパーツに加え，PC本体，ディスプレイ，周辺機器，ソフトウェア，デジタルグッズ，PDA，サプライ用品ほか，ITライフに欠かせない商品群を一挙に取り揃え，ビッグモールに大変身しました。

ザ・モールを一言でいえば「PC関連商品が何でも買えるインターネット商店街」。しかし，ほかのインターネットモールと大きく異なる点があります。それは，「まとめて支払えて，まとめて受け取れる」ザ・モール独自のシステムです。多くのインターネットモールでは，複数商品を注文すると支払いをショップごとに行う必要があり，配達日や時間もバラバラ。でも，「Vwalkerザ・モール」なら何店で買っても支払い・配達は1度でOK。そのうえ，自作パーツ類や一部の周辺機器は，何個買っても送料・代引手数料などは定額なので，注文個数が増えるほどお得というわけです。

商品説明は，/Vmag.OnLineのレビュー記事にお任せ! お買い物をしながら商品の使い勝手やパフォーマンスを知りたくなったときは，ワンクリックで/Vmag.OnLineのコンテンツにジャンプ。もちろん，記事からモールにも直行できます。

毎週水曜日はザ・モールの特価デー!!

11月29日開店

パーツからPC本体,
ソフトまで扱い商品を一挙拡大!

自作パーツ
- CPU ●マザーボード ●ビデオカード
- サウンド ●その他拡張カード類 ●ケース
- HDD ●CD-ROM/R/DVD ●入力デバイス
- クーラー/ファンなど ●その他

ソフトウェア
- ゲームソフト ●ネットワーク
- ビジネスソフト ●ホーム/パーソナル

PC本体
- デスクトップPC ●ノートPC ●サーバー

デジタルグッズ
- デジタルカメラ ●携帯オーディオ
- その他デジタルグッズ

PDA
- Palm ●ハンドヘルドPC(PocketPC)
- その他PDA ●PDA周辺機器

周辺機器
- レーザープリンタ
- 家庭用カラープリンタ(インクジェット/マイクロドライ方式)
- その他プリンタ(昇華型/ラベルプリンタ)
- TA/ルータ ●外付けストレージ
- スキャナ ●その他周辺機器

ディスプレイ
- 液晶 ●CRT

サプライ
- トナー/インク ●切替器
- オフィス家具 ●その他

参加ショップ一覧(順不同)

VIRTUAL T-ZONE.
ムラウチ
でじこん!
マウスコンピューター
ブイショップ
Storm
PCiN秋葉原
高速電脳
フェイス
デュアルコンピュータ
BLESS

参加予定ショップ一覧(順不同)

パソコンショップ BMC
英商
ニッシンパル
クレバリーオンラインショップ
ルーポ
AZ'TECH(アズテック)
FRESH FIELD
・
・
・
・

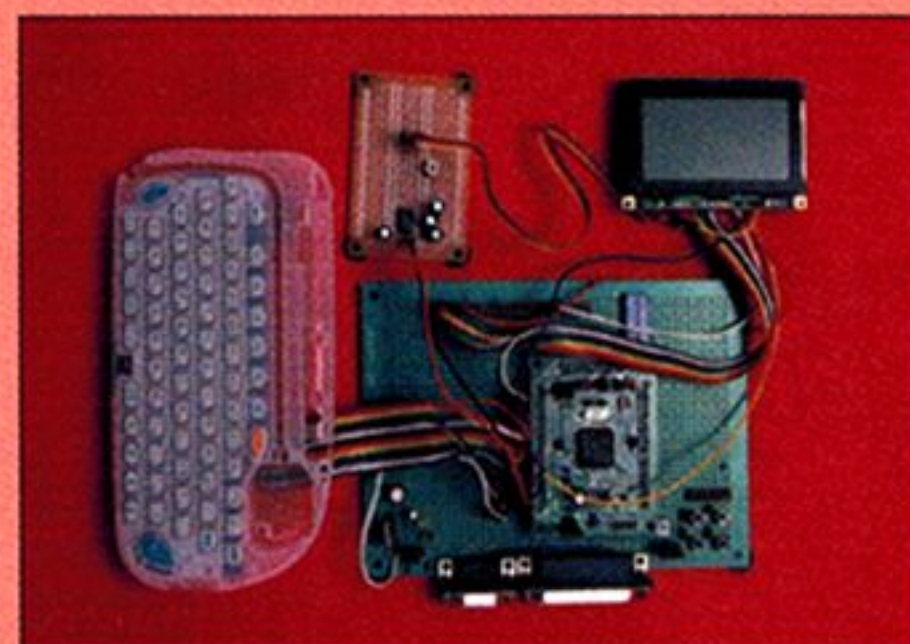
SOFT BANK Publishing

SOFTBANK MOOK

2001春号

Oh!X

CONT

特集　ポケットの中のコンピュータたち

携帯ゲーム機/PDAを作る

彼女と彼女の猫の作り方

Illustratorで主線を書く

特別企画　Xboxと Indremaあるいは Nvidiaの考察

特別企画　オリジナルマシンを作ろう

●特集　ポケットの中のコンピュータたち　～携帯系プログラミング

●Step to the Black Arts

●Oh!X68000

●GraphicLaboratory

©2001 Tetsuya Watanabe

E N T S

<スタッフ>
●編集／植木章夫 ●協力／飯田伸一 石上達也 小村聡 新海誠 田中順子 中森章
野沢絵美 広井誠 福原徹 舩本昇竜 古旗一浩 御木徳高 三津田哲雄
project team DoGA 満開製作所 Visual Houseシブミ サイバーヘッド
●イラスト／岡村直也 岡村祭 高橋哲史 福原徹 Visual Houseシブミ 渡辺哲也
●カメラ／鈴木恒一 ●校正／フィールドアップ
●編集人／來島樹 ●発行人／稲葉俊夫
●DTPデザイン／クニメディア ●印刷／クニメディア

ドラゴンスレイヤー

年末（新年）ですし干支にちなんだものを，と思ったのですが蛇ではよいアイデアが浮かばなかったので"過ぎ行く辰年"というイメージで作ってみました。20世紀のラストボスのドラゴンを退治する，新世紀の騎士といったところです。

RPGを彷彿させる緊張感のない，デザインされた戦い。ちょっと間の抜けた雰囲気を表現してみたつもりなのですが，いかがでしょう？。

表紙 渡辺哲也

インダストリアルCADオペレータを経て現在フリーのCGデザイナー
中部地方で活動中の映像サークルANITEMPの代表
自分の制作した作品を各種上映会，コンテストなどに応募するのが趣味
WFなどのイベントや，専門ショップでビデオも販売中
各種上映会にて上映歴あり
アマチュアCGAコンテスト7，9，10，11回，WavyAwrrd97，98にて受賞歴あり

新海 誠

【URL】http://www2.odn.ne.jp/~ccs50140/ 【E-mail】ccs50140@syd.odn.ne.jp 【制作環境】【ハード】PowerMacG4 400MHz・メモリ576MB・ハードディスク50GB・WACOM ArtPad i600・OLYMPUS D-340L(デジタルカメラ)・230MB MOなど 【ソフト】Photoshop5.0・Shade R4など

はじめに

はじめまして。第12回DoGA CGAコンテストでグランプリをいただいた新海誠と申します。以後お見知りおきを。この記事ではグランプリ作品『彼女と彼女の猫』で使用したテクニックを解説しますが，その前に同作品の手法的な概要に触れておきます。

『彼女と彼女の猫』は約4分40秒の2Dアニメーションです。カット数は字幕のみの箇所を抜かして全74カット。うち，手書きで動画を作画したカットはわずか9カットであり，作画枚数も，1カットにつき3枚から多くて約30枚という少なさです(3DCG部分も含めれば動画シーンはもう少し多くなります)。『彼女と彼女の猫』では，作品中の多くの時間を静止画(または極めて限定された部分的なアニメーション)が占めています。このような構成をとっているのは，とりもなおさず制作時間の短縮が目的でした。静止画を多用する分，ナレーションと効果音を活用して「動かさずに済む」演出を徹底して行っています。

同時に，このような演出面での工夫のほかにも，同作品は作画にあたってデジカメ画像を徹底的に利用することで省力化を図っています。全カット中40カット以上にデジカメ画像をもとにした作画を行うことで，省力化と同時にクオリティ面でもそれなりの成果を挙げることができたと思っています。『彼女と彼女の猫』のダイジェストムービーをCD-ROMに収録しましたので，原稿とあわせてご覧ください。

本稿では，『彼女と彼女の猫』の制作手法の要であった「デジカメ画像をもとにイラストを描く」方法を解説します。題材は本稿のための新作イラスト「扇風機」です。そのあとで『彼女と彼女の猫』から3シーンを例に取り，アニメーション化までの流れを解説します。

実践・「扇風機」

では，実際に作成してみましょう。テーマは夏，題名は「扇風機」。モチーフとなる女の子をデジカメで撮影し，その写真をもとにイラストを描いていきます。

step 1 写真を撮る

まずは写真が必要なのですが，撮る前に完成形をイメージしておきましょう。写真をもとにするので構図の明確なイメージまでは必要ありませんが，描きたいシチュエーションだけでもはっきりさせておくとあとの作業がスムーズになります。

今回は，(1)扇風機の風に当たりながら，携帯電話で話している女の子，(2)涼しげなフローリングの床，(3)傍らにスイカというシチュエーションを想定しました。このイメージをもとに撮った写真が図1です。

撮影にあたっての注意点は，「トレースしたい素材はなにか」を明確にしておくことです。今回はとにかく携帯で話す女の子が写っていればいいわけですから，背景には特に気を使っていません。また，撮影時にちょうどいい扇風機がありませんでしたので，後に描き換えることを念頭にモデルの女性には椅子に手を載せてもらっています。

また，輪郭トレースのための情報がなるべく多くほしいのなら，フラッシュを使用したほうがよい結果を得られる場合が多いでしょう。逆に，その場の陰影も含めた雰囲気をイラストに反映したい場合は，フラッシュの効果が邪魔になることもあります。ちなみに今回はフラッシュは使用していません。

撮影した画像は基本的にトレースの下地とするだけで最終完成画像には反映されませんから，デジカメの解像度は問いません。今回は手持ちのデジカメの最大解像度である1280×960ピクセルで撮影しましたが，実際のトレースを行う段階(step 3)では横3000ピクセルほどまで拡大しています。デジカメの機種もなんでも構わないのですが，参考までに私の環境を紹介しておきます。

デジカメはオリンパス光学工業のD-340L，画

図1 デジカメで撮影した元画像。これをトレースしてイラストを作成していく

図2 プロペラを囲む細いフレームは線形状に沿った掃引を利用して作成

図3 フローリングの床はこのような基本パターンを敷き詰めて構成。ひとつひとつの板は閉じた線形状を掃引しているだけ

素数は130万です。画素数についてはどの程度まで写真のディテールをトレースに利用したいかによりますが，個人的にはこの程度のスペックの機種でも十分だと思っています。コンピュータへの画像取り込みはロジテックのUSB接続のカードリーダーを使用しています。

step 2 3Dモデリングとパースあわせ

背景の床と扇風機は，3Dソフトで作成することにします。もちろん手描きでも構いませんが，3Dソフトを使用すれば写真とのパースあわせが非常に楽に行えます。私はShade R4を使用していますが，パースあわせの際に必要になるテンプレート取り込み機能さえあれば，どの3Dソフトでも問題はないでしょう。Shadeのコマンドで説明しますが適当にお手持ちのソフトの機能と置き換えてお読みください。

扇風機とフローリングの床のオブジェクトを作成します(図2・図3)。最終的には手で輪郭をトレースしますので，細部に凝る必要はありません。扇風機の首の部分には回転ジョイントを仕込み，上下左右に首を振れるようにしておくとレイアウトを決めるときに便利です。

オブジェクトが完成したら，撮影したデジカメ画像をテンプレートとして読み込み，パースペクティブをあわせます。Shadeの場合パースペクティブビューにはなんのグリッドも表示されませんので，まずは下準備として，開いた線形状で簡単なグリッドを作っておきます(図4)。グリッドを入れたパートの名前の頭に`をつけておけば選択ができなくなりますから，作業の邪魔になることもありません。

準備ができたら，File→Load Template→Pres...でパースペクティブビューに撮影したデジカメ画像を読み込みます。Shadeのテンプレート画像は拡大縮小ができませんから，あらかじめ作業するパースペクティブビューのサイズにあわせて読み込む画像は縮小しておきます(Shade R3以前ではビューごとにテンプレートを読み込むことができませんので，画面表示をパースペクティブビューのみにする必要があります。1999年春号の森川さんの記事で詳しく解説されています)。今回の画像は畳のラインがパースをあわせるのに都合がよいので，これにグリッドが沿うようにします。ShadeでのCameraZoomの値は40mmとしました。この状態で扇風機のオブジェクトを配置すれば，テンプレート写真とオブジェクトのおおよそのパースは一致することになります(図5)。

ここまでで，ひとまずスキャンラインで扇風機のみ仮レンダリングして，Photoshopにてデジカメの元画像と合成してみます(図6)。パース的な違和感がなければOKです。

step 3 Photoshopでの人物の輪郭トレースと着彩

まず，イラストの最終出力サイズにあわせて元画像を拡大します。今回は印刷媒体への出力を考え，約3000×2000ピクセルで作成しています。まず，トレースした線を見やすくするために元画像を白っぽく半調にします。白く塗りつぶしたレイヤーを50%程度で重ねてもいいでしょうし，トーンカーブなどで直接元画像を加工してしまってもよいでしょう。これをブラシツールでひたすらトレースしていきます(図7)。ある程度大きなキャンバスですので，ブラシサイズは5ピクセルのものを使っています。

この手作業でのトレース作業の出来不出来が，最終的なイラストの仕上がりに決定的に影響します。単純に写真の輪郭をトレースするといっても，まずどのエッジを輪郭線として認識するのかという問題があります。また，どの部分のディテールを描き込んでどの部分を簡略化するかという問題もあります。また，顔や指，服のしわ，髪の毛のラインの引き方など，手描きの線ならではのデフォルメを施す必要もあるでしょうし，モデルの女性のスタイルがどれほど理想に近くても，イラストにする際には首や肩の幅，髪の毛のボリュームなどの調整が必要になる場合がほとんどです。

このあたりの処理は絵柄の好みにもよりますし，一般化しての説明が非常にやりにくい部分なのですが，参考までに手の処理(図8・9・10・11)と足の処理(図12・13・14・15)を挙げておきます。図9・13をご覧いただけばわかるとおり，写真の中で輪郭線らしきものをそのままトレースしても，ほとんどの場合はぱっとしないものになってしまいがちです。個人的には，曖昧さをなるべく排除すべく，曲線部分も直線の連続で描く，というくらいの気持ちでやっています。

図16がトレース完成画像です。髪の毛を風になびかせているほか，顔や指先の処理，携帯のディテール，首や背中のラインなどを，トレースの段階でアレンジしています。写真をもとにしていることで全体のデッサンは狂いようがありませんので，「忠実にトレースする」ことよりも「写真の印象を再現する」ことを目標に作業を行うとうまくいきやすいと思います。

ちなみに元画像に単純にPhotoshopの輪郭検出フィルタをかけるとこうなります(図17)。

続けて着色に入ります。今回は写真の陰影がよい感じでしたので着彩の参考にしていますが，着

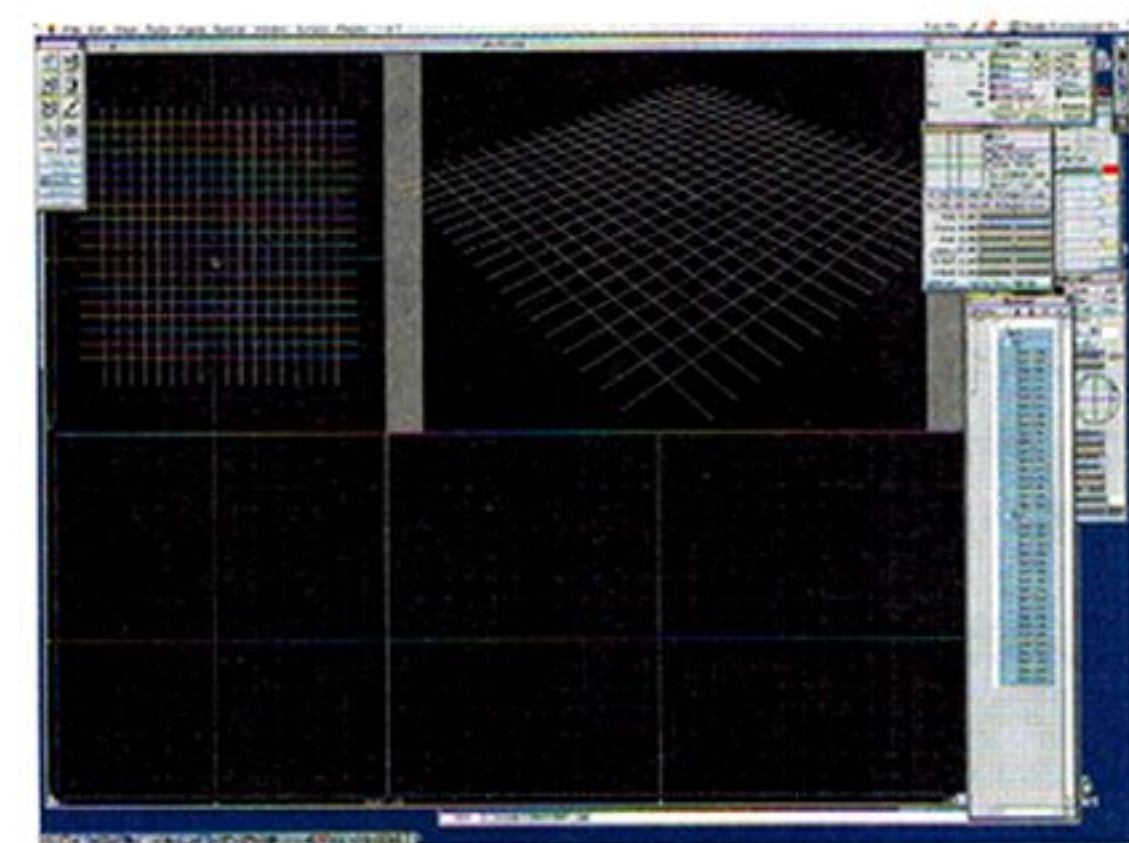



図4 開いた線形状でパースあわせのためのグリッドを作成

図5 テンプレート画像にグリッドのパースをあわせ，扇風機のオブジェクトを配置

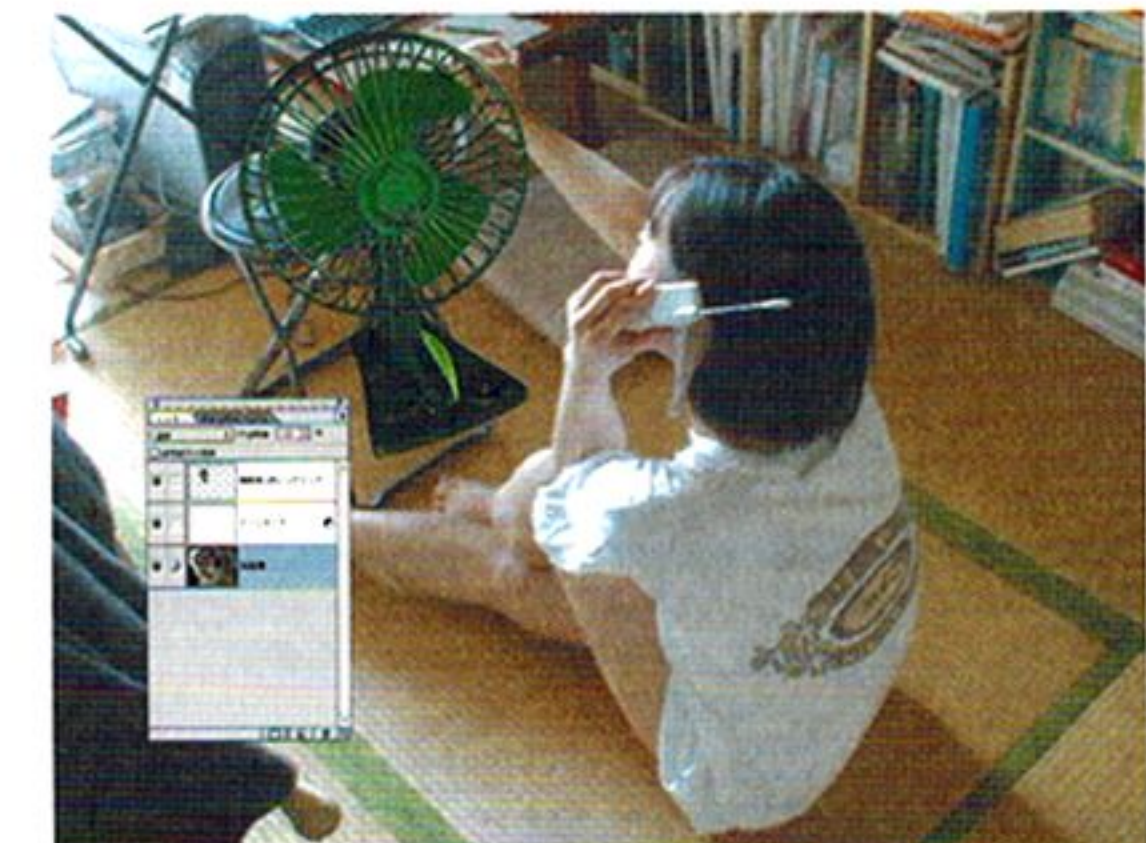



図6 仮レンダリング画像を元画像に合成してみる

図7 Photoshopでの輪郭トレース。タブレットがあったほうが便利

図8 元写真。これだけでは，どこを輪郭とすべきか判然としない

図9 とりあえず素直にトレース。メリハリに欠け，イラストとはいいがたい

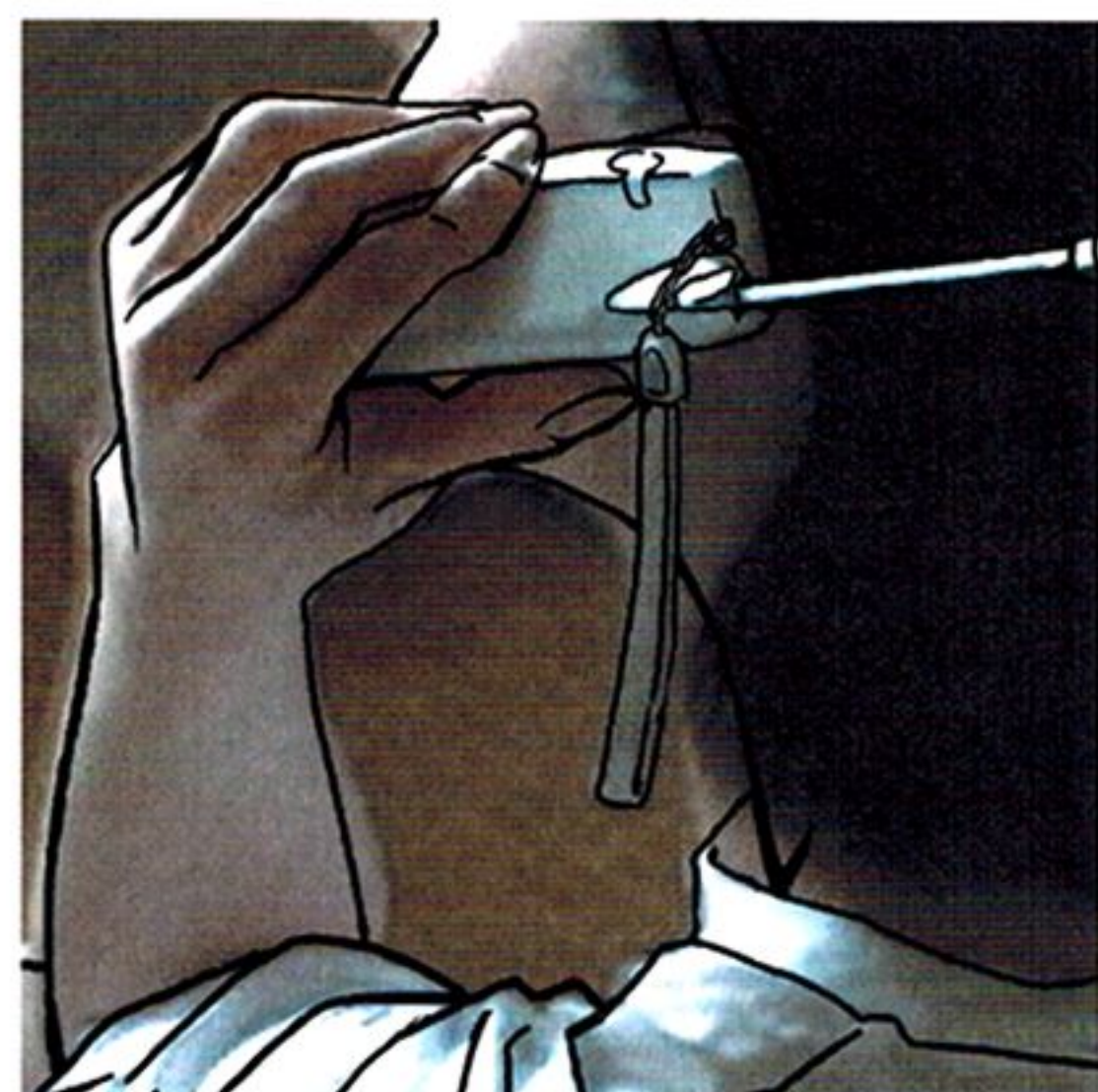



図10 写真はある程度アレンジしてトレースしていく

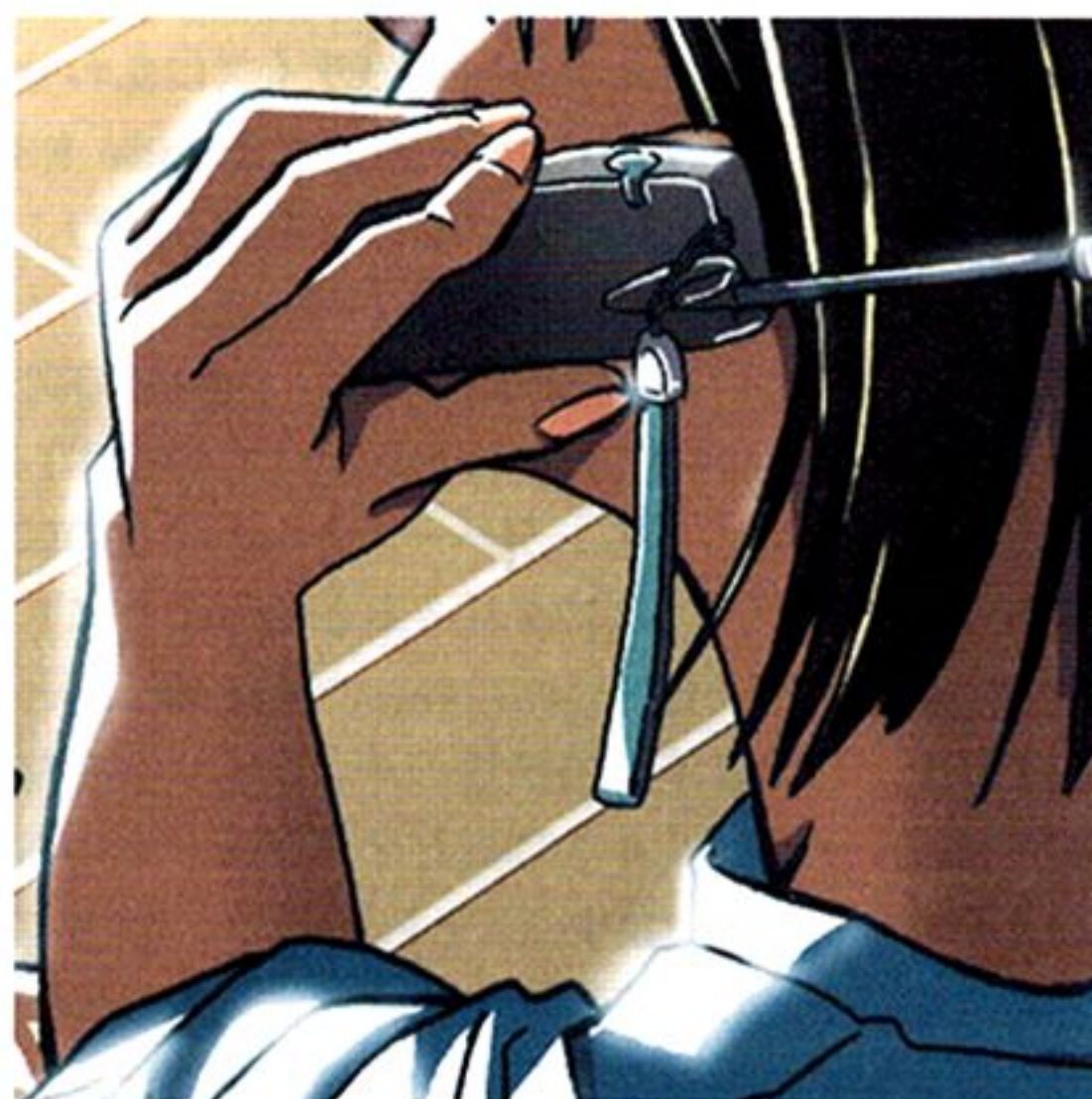



図11 最終的には陰影で立体感を補助するので，写真に引っぱられてあまり輪郭を複雑にしないほうがよい

図12 しつこく元写真。足の例

図13 素直にトレース。やはり形が曖昧になる

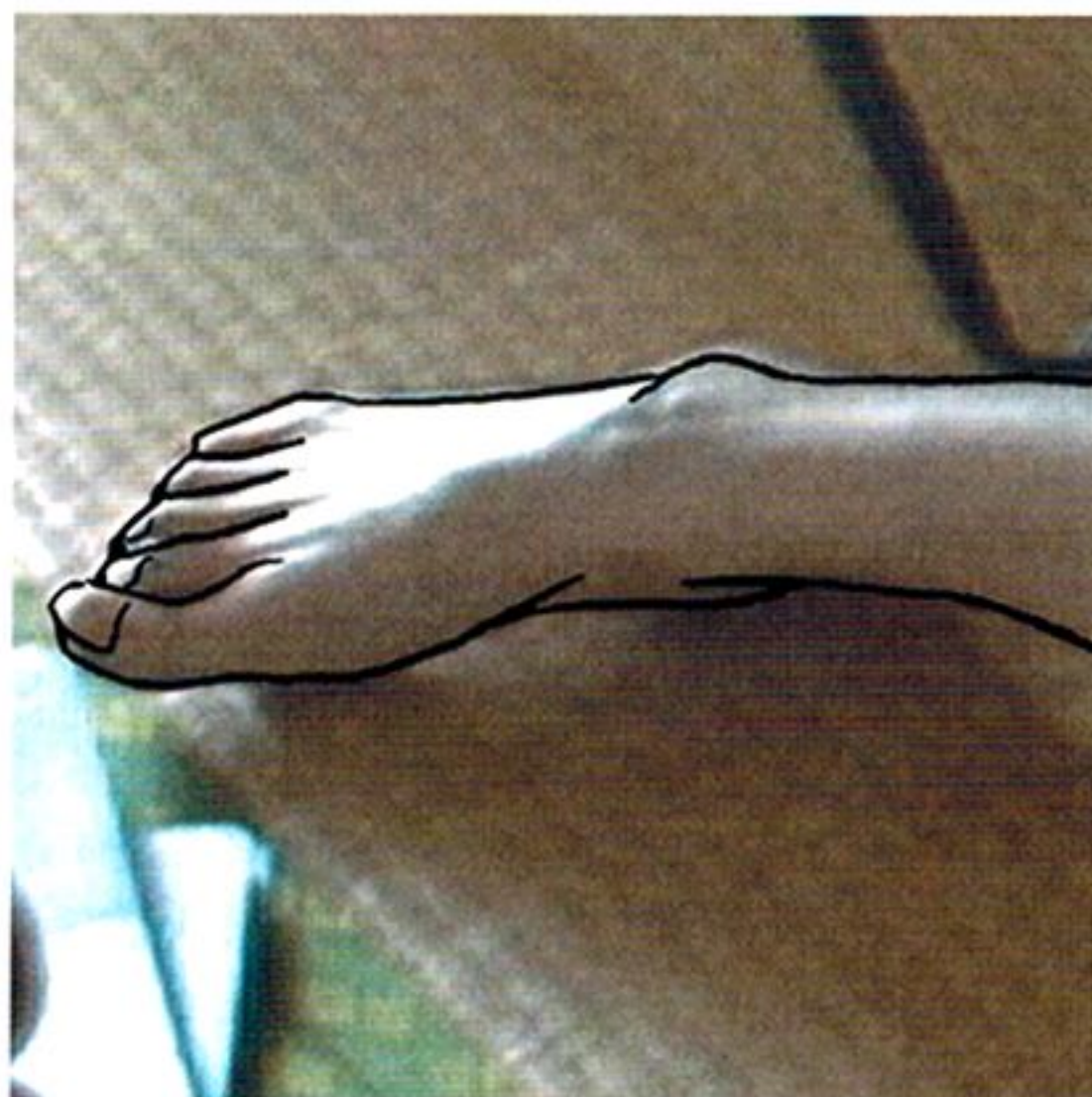
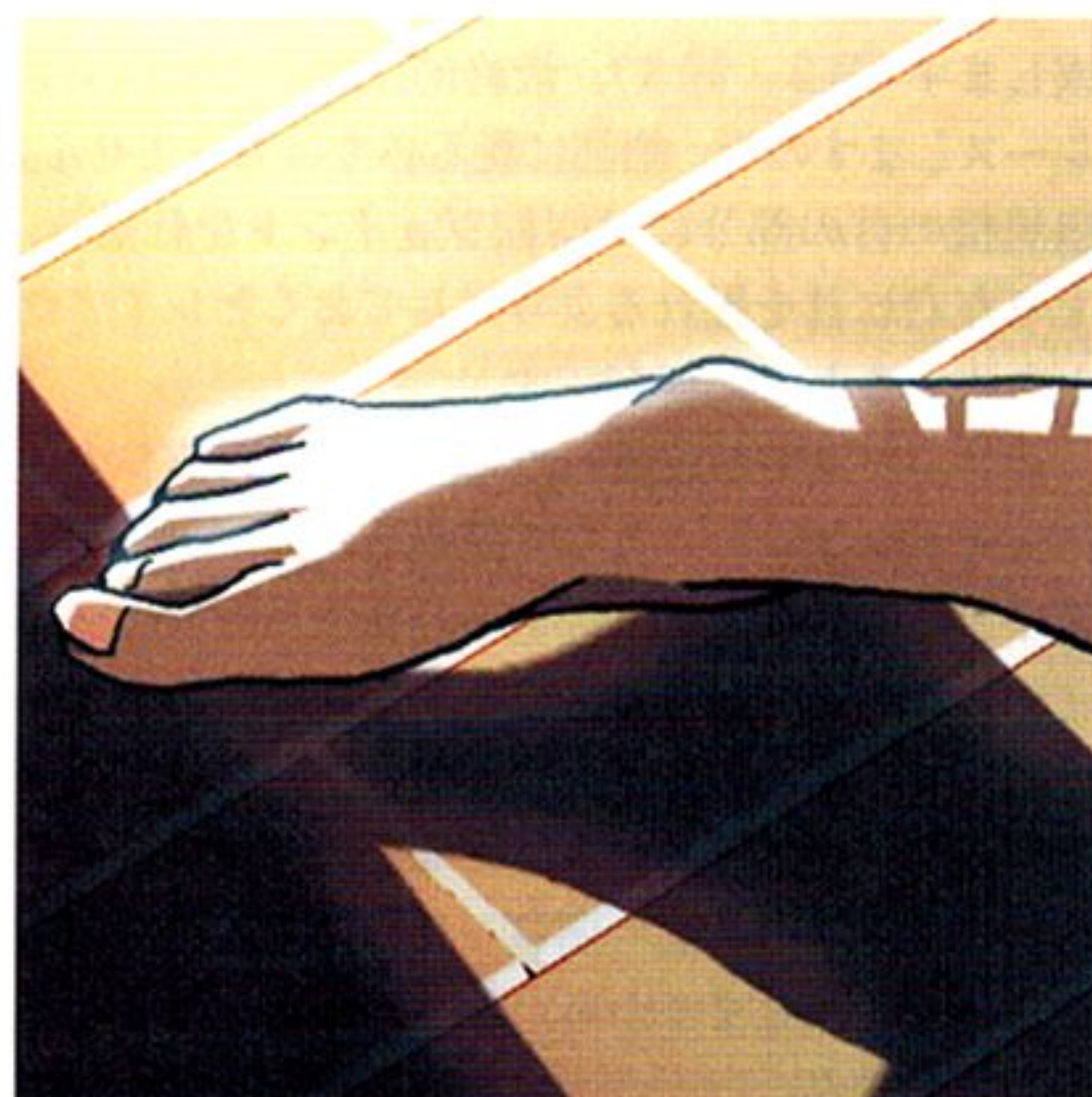



図14 必ずしも写真に忠実でなくともよい

図15 完成画像

図16 人物の輪郭トレース完成。気持ちのいい写真と気持ちのいいイラストは違うので，部分的にはアレンジも必要

彩については元写真を無視したほうがうまくいく場合も多々あります。端的にいえば，現実の陰影はそのままイラストにするには複雑すぎるからです。今回のイラストからは少し外れますが，『彼女と彼女の猫』では資料として撮りためておいた写真を素材として使用したカットも多かったので，特に人物の陰影についてはほとんど元写真を無視しています。例として1カットを挙げておきます（図18・19）。

着彩の技法そのものに関しては特に複雑なことはやっていませんが，とにかくベース色，シャドウ，ハイライトなど，要素ごとに徹底してレイヤー分けをすることにしています（図20・21・22・23）。こうすることによっていつでも修正が利きますし，【step 5】で説明しているような，塗りのタッチそのものの変更にも対応できます（当然メモリを消費するというデメリットもありますが，昨今のメモリ状況を考えれば十分に現実的だと思います）。

step 4 3Dオブジェクトの合成と輪郭トレース・背景の作成

扇風機のイラストを仕上げます。「輪郭を手作業でトレースしたい」「プロペラを回転しているよ

図17 比較することの意味はないが，Photoshopの輪郭検出フィルタを元画像にかけるとこうなる。このような用途にはあまり役に立たない

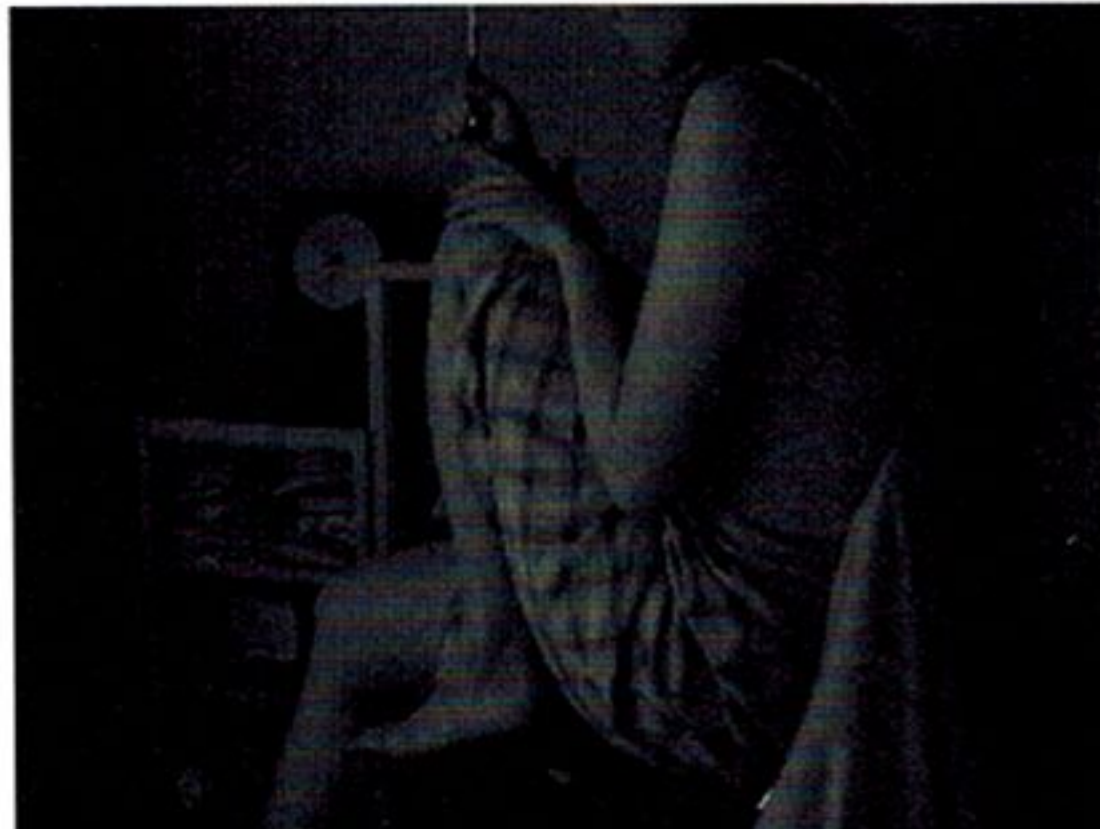



図18『彼女と彼女の猫』で使用した，資料用デジカメ写真

図19 図18の実際の使用カット。手の大きさなどのバランスをアレンジしているほか，陰影づけに関しては元画像を完全に無視している

図20 人物着彩過程1。肌・髪・Tシャツ・携帯の4パーツを別々のレイヤーに塗る

図21 人物着彩過程2。4パーツごとにレイヤーのグループ化を使用して影を載せる

図22 人物着彩過程3。さらにハイライト，エアブラシで微妙な陰影，濃い影などを追加。髪の毛のハイライトのみ覆い焼きで重ねている

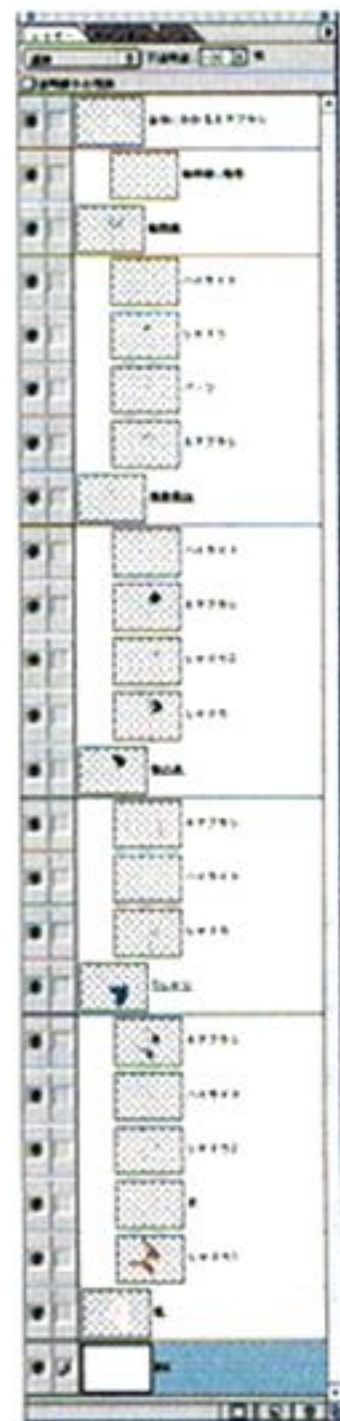
うに見せたい」という目的に沿うため，ベース・プロペラ・フロントカバーで別々にレンダリングしておきます(図24)。これをもとにやはり輪郭を手描きでトレースし(図25)，さらに人物と同じ手法で着彩していきます。回転しているプロペラの表現は，レンダリング画像に回転の放射状ぼかしをかけ，そのレイヤーを回転させたものを複数枚重ねることでお手軽に実現しています(図26)。ただしこれだけではメリハリに欠けますので，手描きで適当なハイライトを重ねて完成です(図27)。

ここまでで，メインのモチーフとなる人物と扇風機は完成です。Shadeでのパースをあわせたフローリング画像(図28)と合成して，全体を仕上げていきます(図29・30・31)。

図31の状態で一応の完成としてもよいのですが，さらに夏らしい小物を追加してやることにします。スイカ，ペットボトルと読みかけの文庫，脱ぎ捨てたジーンズを描き加えました(図32)。これらの小物はアドリブで描きましたが，もちろんほかのデジカメ画像を配置してトレースしても，または3DソフトでオブジェクトIを作成しパースを合わせた空間に配置してもOKです。小物も人物と同じ手順で着彩を行います(図33・34)。追加した小物の影を床に落とし，これですべての要素がイラストに含まれました(図35)。

図23 人物のレイヤー構成

step 5 仕上げ・水彩調画像へのアプローチ

ここから先の補正はほとんど趣味の問題ですが，参考までに言及しておきます。図36が最終完成画像です。図35との違いは，

①ハイライトの明るさを強調する光のふくらみをエアブラシで吹いている

②日向と日陰の温度差を強調するため，影のエッジに色を流している

③トーンカーブやカラーバランスの微調整

④女の子の顔に焦点をあわせ，ほかをわずかにぼかしている。同様に，ボケに応じてノイズをのせてなんとなく実写っぽく加工

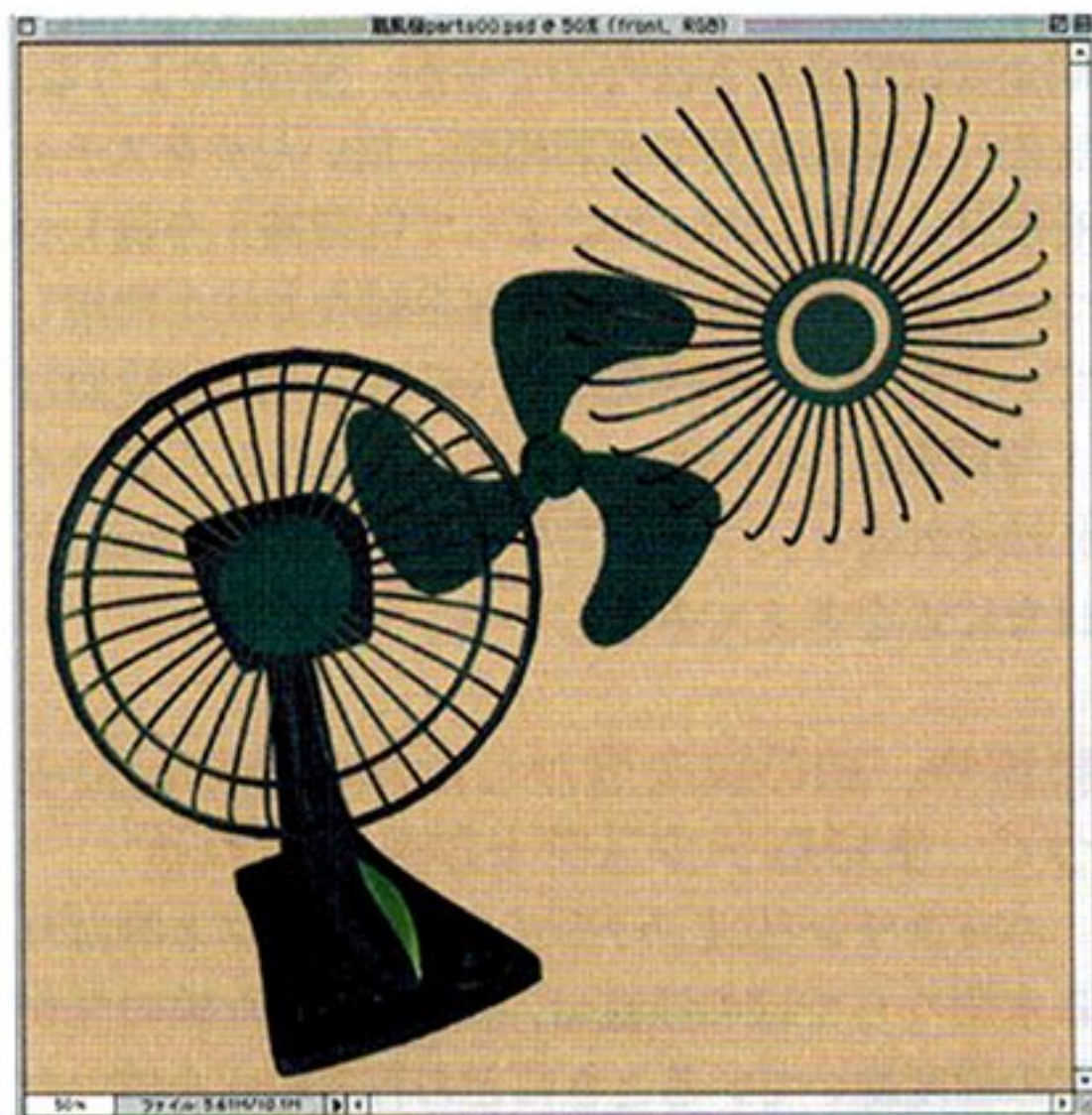



図24 扇風機のオブジェクトを3つに分けて別々にレンダリングしておく。ちなみに影("shade"ではなく"shadow"のほう)は手でつけるので，スキャンラインでのレンダリングでよい

図25 例によって手作業で輪郭をトレース

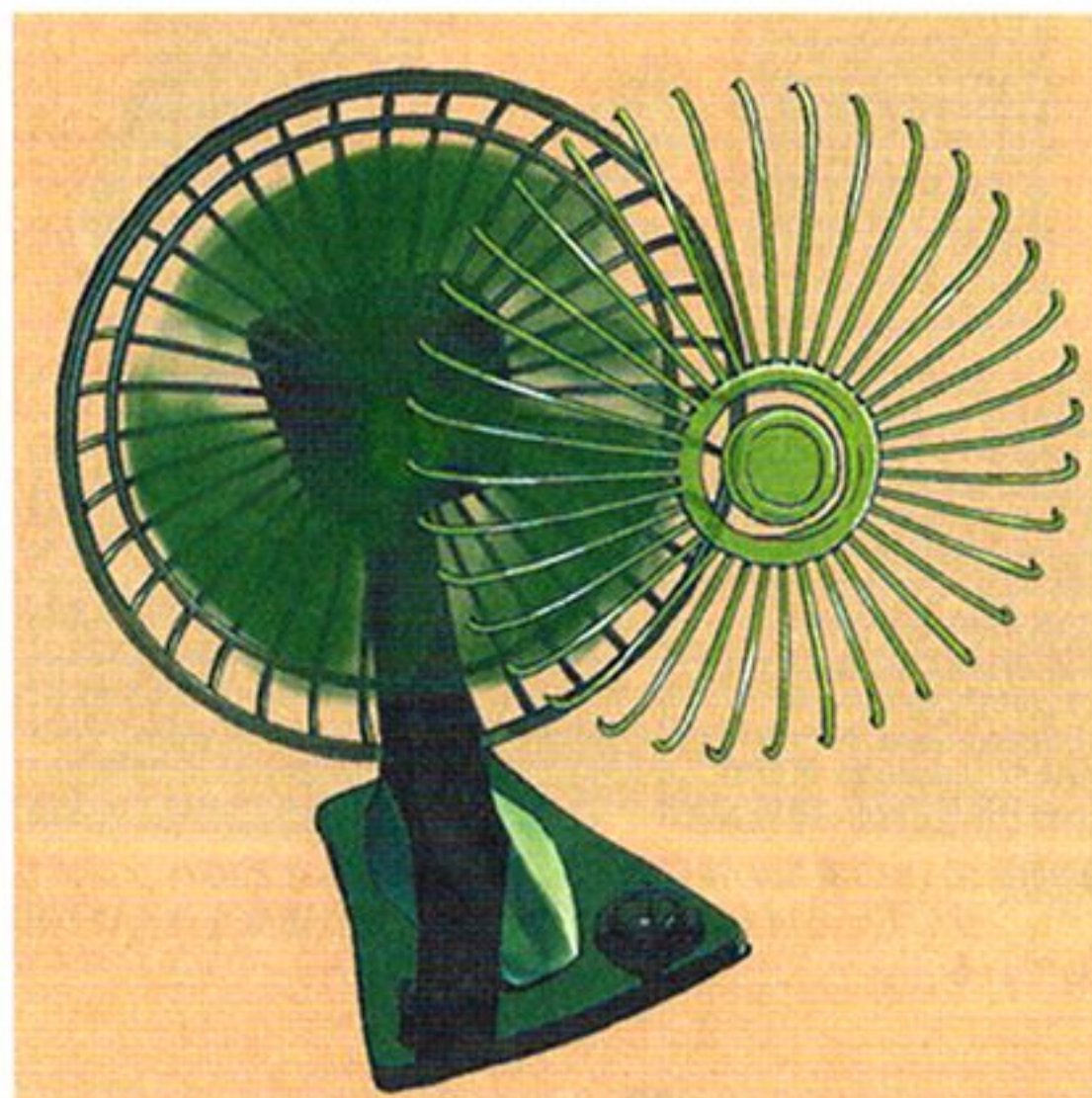



図26 人物と同様にパーツごとに着彩していく。回転するプロペラの表現は放射状ぼかしを使用

図27 それらしくハイライトやシャドウを入れ，完成

図28 フローリング部分のみのレンダリングも行っておく

図29 人物と扇風機をフローリングの床に配置。ちなみに人物・扇風機ともに，線画・着色部分以外は透明をキープしているので，合成の際の切り抜き作業は必要ない

図30 扇風機の影を描く労力を省略するため，Shadeのレイトレーシングで影を落とした画像も作成しておく。レベル補正とアルファチャンネルを使用して影の部分のみ切り抜いて使用する

図31 図29に人物と窓枠の影を描き，さらに図30の扇風機の影を合成する。人物の足にも図30の影の一部を落としてやる。さらに床と影には明暗や色彩のメリハリをつけて画像の中に温度差をつける。明暗をレイヤー別にしておいてやるとこのあたりの作業が楽

図32 アドリブで小物を追加

図33 スイカ。かなり適当

図34 ジーンズ。ノイズフィルタや指先ツールなどを使用してそれっぽいテクスチャを載せている

図35 ひとまず完成。ちなみにフローリングの床は輪郭のトレースなどは行わず，レンダリング画像に明暗の加工をかぶせただけ

などです。

最後に，おまけとして図36の完成画像をもとに水彩調に加工したイラストも掲載しておきます(図37)。一見して，ペイントソフトでのいわゆる「水彩調」とは一線を画した表現が見て取れると思います(好き嫌いはあるでしょうが)。これも種を明かせば簡単な方法なのですが，図38のような，実際に水彩絵の具で水彩用紙に描いたテクスチャを取り込んで，パーツによって色調補正を施しつつレイヤーで重ねるというお手軽な方法を使用しています(図36から図37への加工の所要時間は1時間弱)。お手軽とはいえ，水彩らしく見せるにはそれなりにコツがあります。水彩画の特徴とはなにかを考えてみましょう。

紙質や，絵の具を溶かした水の量などによって，多彩なテクスチャが生成される

バックラン(にじみ)やブレンディングと呼ばれる水彩ならではの技法は，「紙に塗った絵の具がどの程度乾いているときにどの程度の水を溶いた色を流し込むか」など，非常にコントロールが難しく偶然に頼った感もある技法である分，非常に特徴的なテクスチャが生成されます。

これらの手法をソフトウェアで再現するアプローチがどの程度存在しているかは知りませんが，「筆・絵の具・水」という多様なデバイスを自在に扱って描く実物の水彩と同等の表現をコンピュータで行うことが，相当に難しいであろうことは想像にかたくありません(Painterなどではある程度しっかりと水彩をシミュレートしているのかもしれませんが，使ったことがありませんし，目を引くような水彩の作例を見たこともあまりありません)。

そこで，手っ取り早くそれらの水彩独特のテクスチャを実現する手段として，図38のような実物をスキャニングして切り貼りする，という方法が非常に有効になります。

バックラン，ブレンディング，ブラッシュマーク(筆の跡)，ハードエッジ(乾いた絵の具の縁の部分)など，実際の水彩の技法ごとのテクスチャ

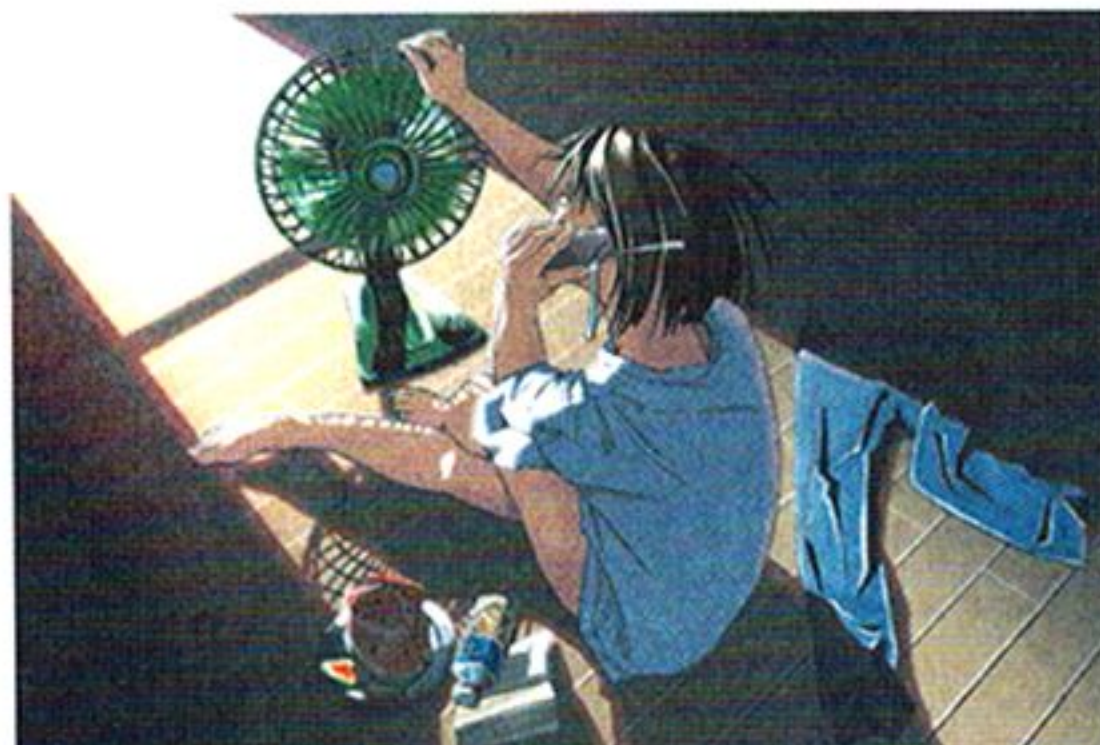



図36 最終的な完成画像

図37 図36をもとに，実際にスキャニングした水彩テクスチャを利用して透明水彩調の表現を行った

図38 実際に水彩用紙に水彩絵の具でテクスチャを作成した。スキャニング後にPhotoshopでシームレスなテクスチャに加工している

図39 完成ムービー

図40 デジカメでの元画像

図41 輪郭をトレースしてイラスト化。ビニール袋や床面は写真のテクスチャをそのまま利用して省力化を図っている

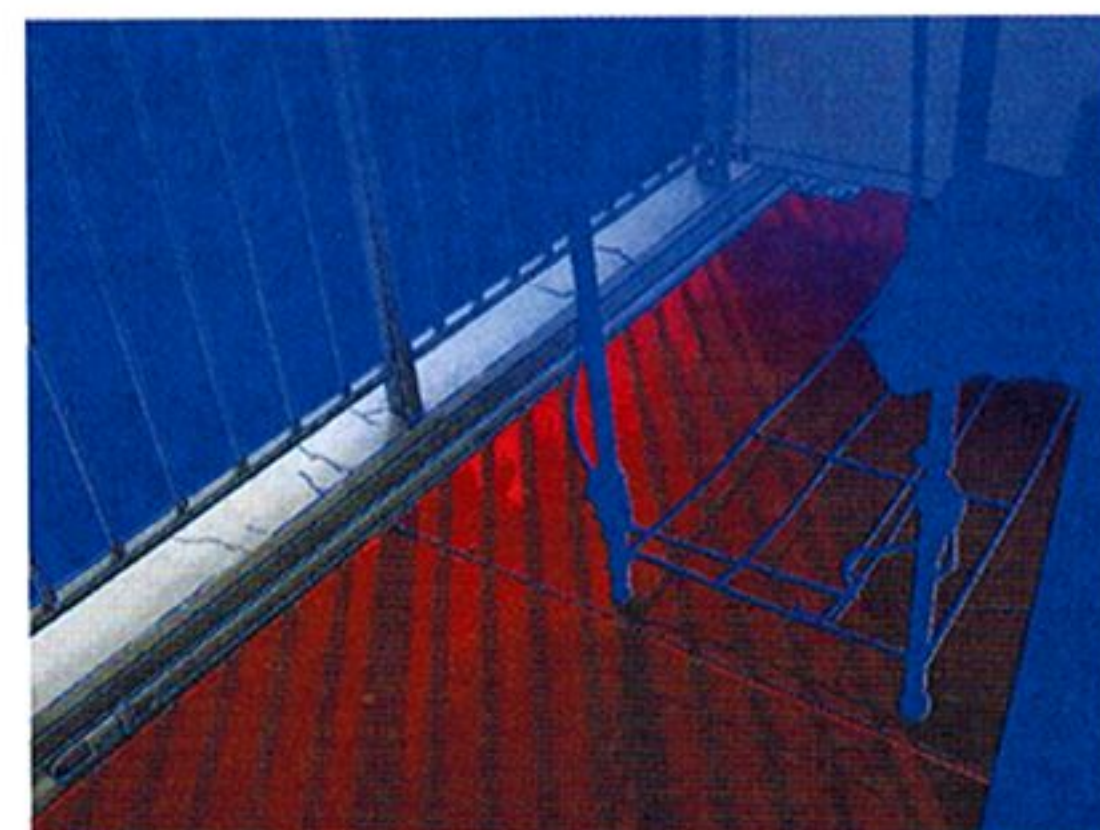



図42 波紋を床面だけに発生させたいので，マスクを作成する。赤が波紋を発生させたい部分。青は波紋を隠す部分

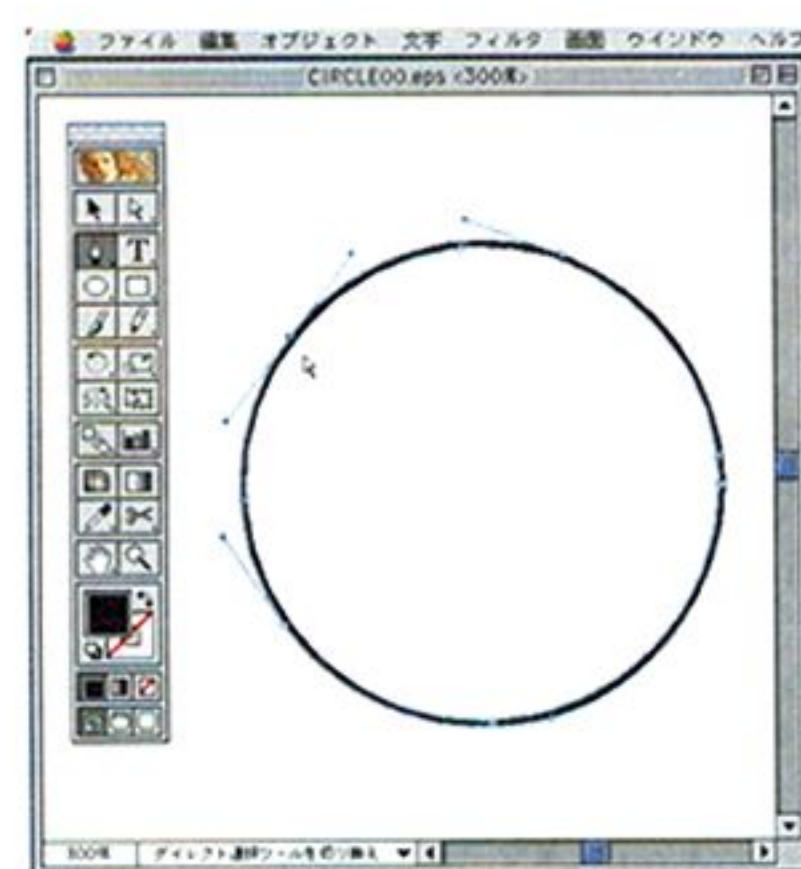



図43 波紋のもとを作る。拡大縮小の劣化を抑えるためIllustratorで円を作成したが，別にPhotoshopでも可

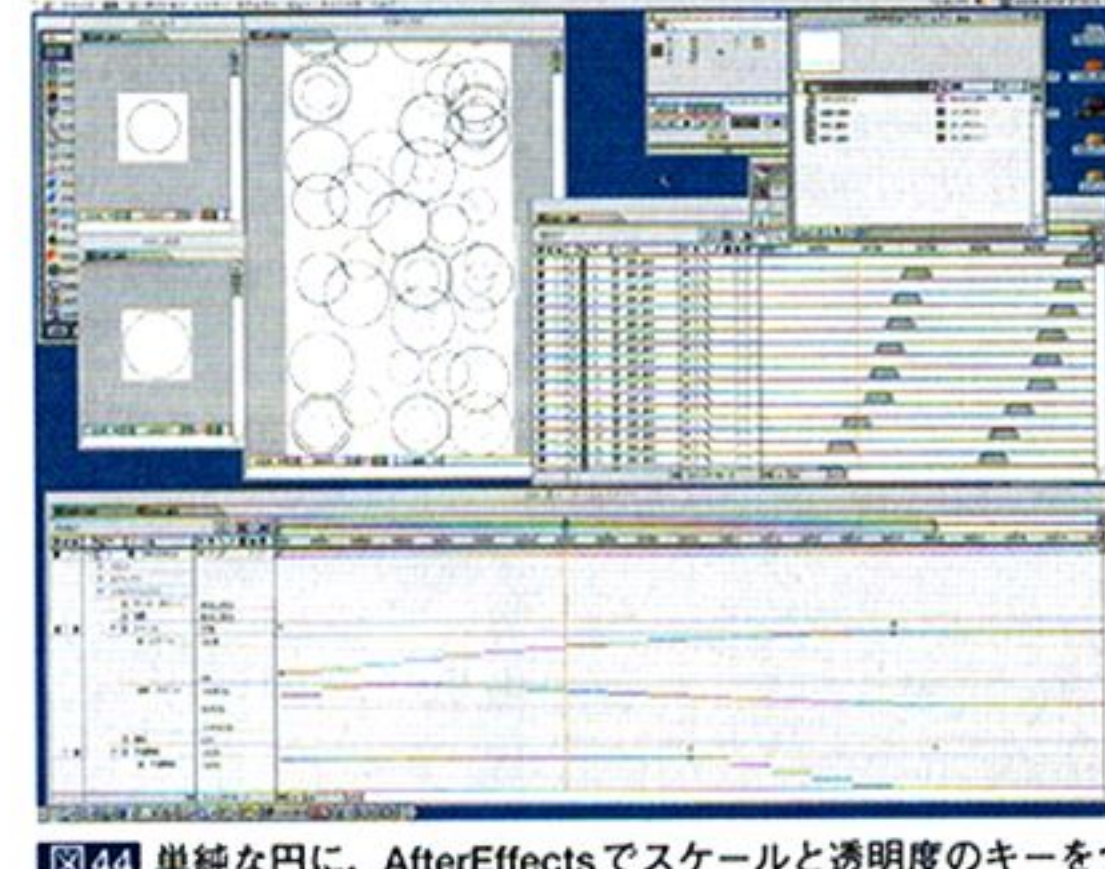



図44 単純な円に，AfterEffectsでスケールと透明度のキーをつけて，円が広がりながら消えていくアニメーションを作成。それを手作業で時間軸をずらしつつ複数並べていく

を用意しておけば，さらに本格的な表現も可能でしょう。

透明水彩では，基本的に白は紙の塗り残しで表現する

透明水彩に特徴的な透明感やきらめくようなハイライトは，「ハイライト部分には絵の具を使わずに紙を白いままに塗り残す」という手法によるところが大きいといえます。ガッシュなどの不透明絵の具と異なり，透明水彩では重ね塗りでは白を表現できません。そこで，コンピュータ上で水彩を再現する際にも，基本的にはハイライトは白のまま残すようにすることでより水彩らしさを強調できます。

今回のイラストは，最後までハイライトやシャドウをレイヤーで分離しておくことで，「シャドウ部分にのみ水彩テクスチャを貼る」という加工が容易にできます。とはいえ，それほど厳密にすることもありませんので(実際のアナログ作業でも透明水彩にガッシュの不透明な白を併用することは多々ありますし)，今回のイラストでは人物を除き，扇風機などの小物では単純にハードライトで全体の上から水彩テクスチャを重ねています。

『彼女と彼女の猫』でのデジカメ画像使用例

第12回DoGA CGAコンテストでグランプリをいただきました『彼女と彼女の猫』の制作においても，デジカメ画像をフルに利用しています。基本的には上記の手法でデジカメ画像をイラスト化して使用していますが，シーンによってはそのようにして作成したイラストをさらにアニメーションにしています。以下で，『彼女と彼女の猫』から3シーンを例にとり，それらがどのような手順で制作されたかを解説します。

なお，『彼女と彼女の猫』の制作時のマシン環境についても簡単に言及しておきます。当時のマシン構成はPowerMac7600/120＋G3-233MHzボード，256MBのメモリ，ハードディスクは8GBほどでした。決して快適な環境とはいえませんでしたので，作品は一貫してグレースケール，15fpsで制作しました。また，主な使用ソフトはPhotoshop5.0，Lightwave5.6，AfterEffects3.1(途中で4.0にバージョンアップ)です(補助的にIllustrator8.0とShade R3も使用)。

ベランダに降る雨の波紋のアニメーション

約3.5秒，SEとして雨の音，室内の留守番電話の音が鳴っているシーンです(図39)。波紋は一見すると3Dソフトで作成したように見えるかもしれませんが，AfterEffectsのマスクとPhotoshopのパッチを駆使して作成しています(図40～47)。

アパートのドアのアニメーション

女性が部屋の外に出ていった直後のシーンです(図48)。省力化のための演出として，ドアが閉まる「バタン」というSEの直後にこのシーンがINしてきますので，出入口のドアそのもののアニメートは必要ありませんでした。ただし，余韻を残すために手前にあるリビングのドアとその影がアニメートします。Lightwaveのセルシェーディング

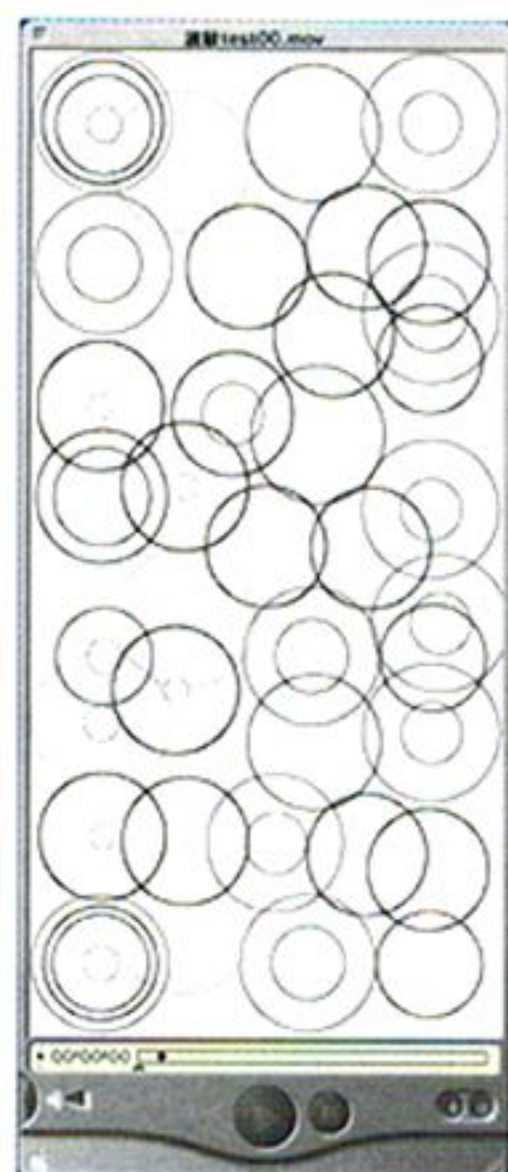



図45 完成した波紋の基本ムービー

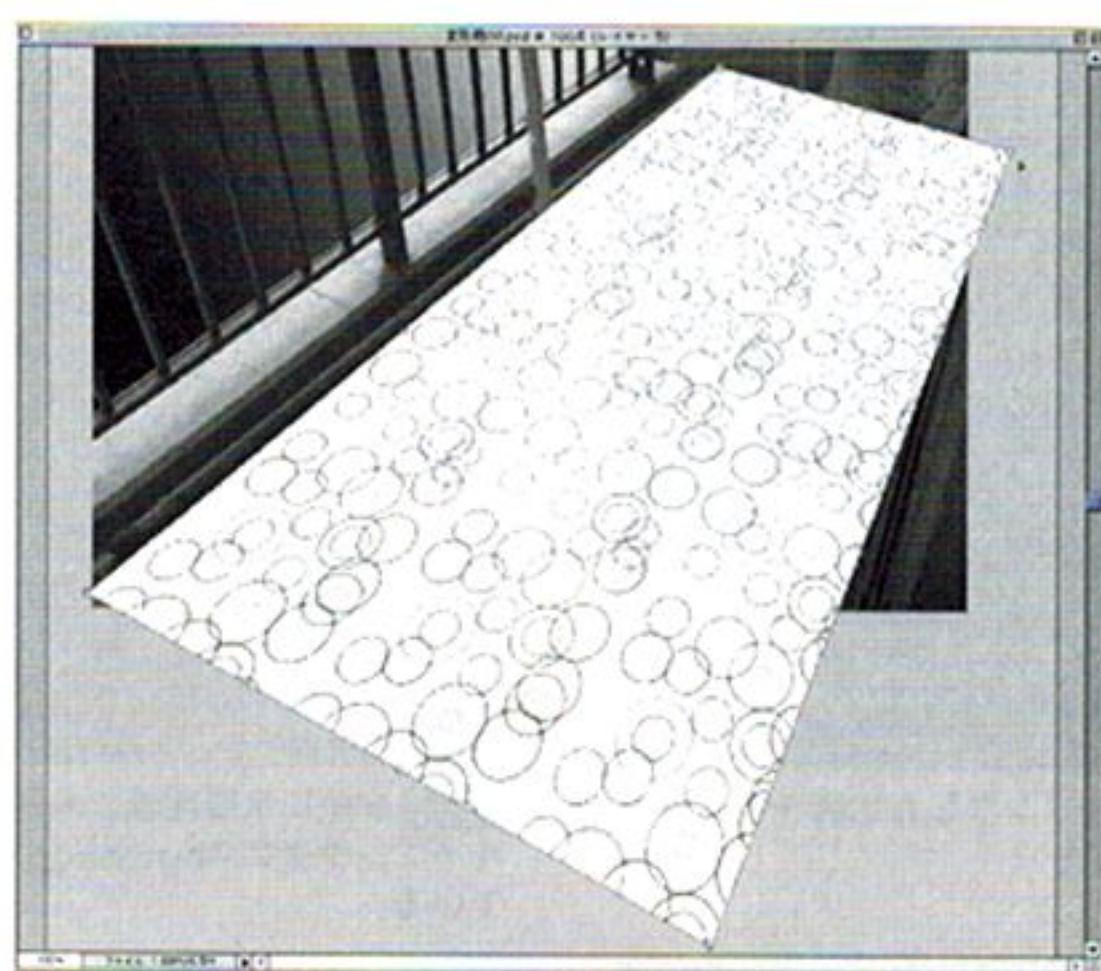



図46 図45で作成した波紋の基本ムービーを複数枚並べ，Photoshopシーケンスとして書き出す。それをPhotoshopで図のように変形させ，その過程をアクションとして記録。バッチ処理を行えば，パースのついた波紋の連番シーケンスができあがる

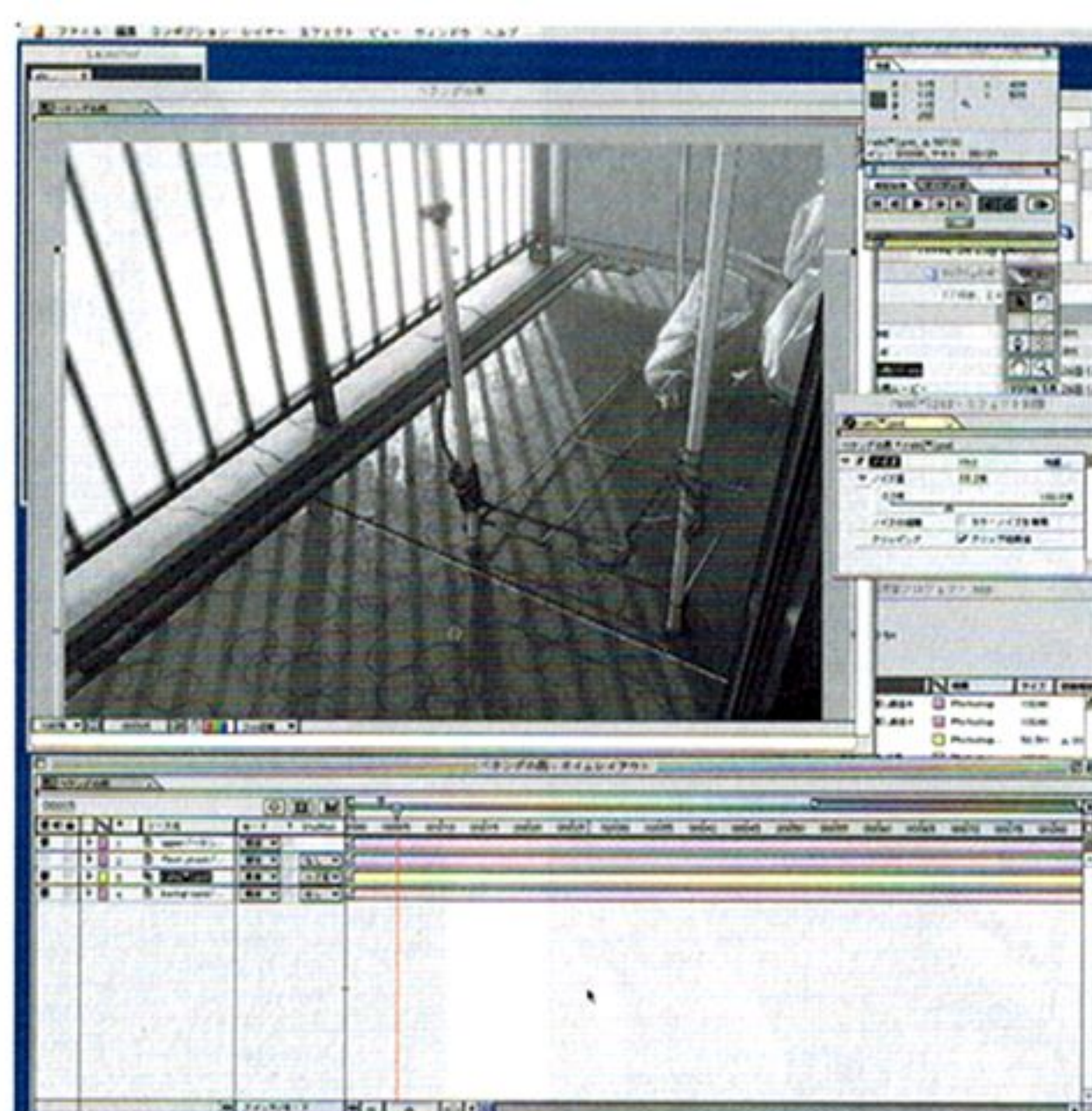



図47 図42で作成したマスク情報をAfter Effectsでトラックマットとして使用し，背景イラストに波紋シーケンスを合成して完成

図48 完成ムービー

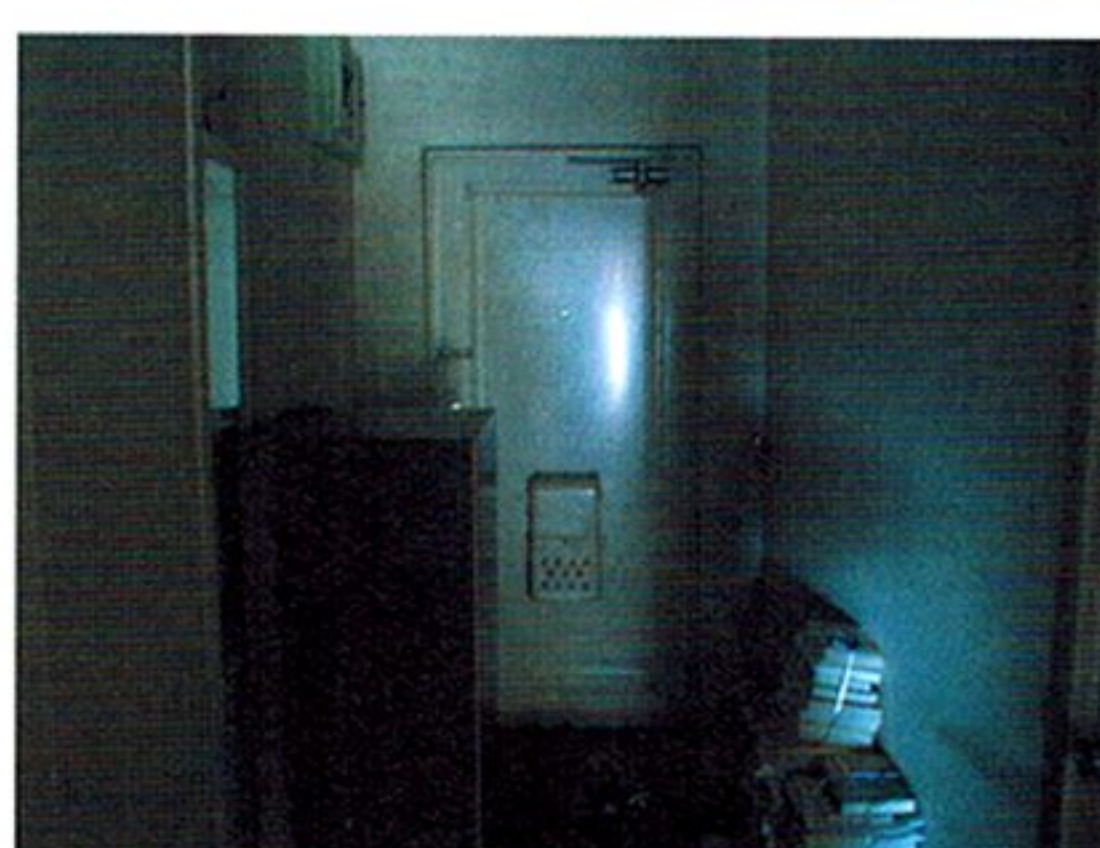



図49 デジカメでの元画像

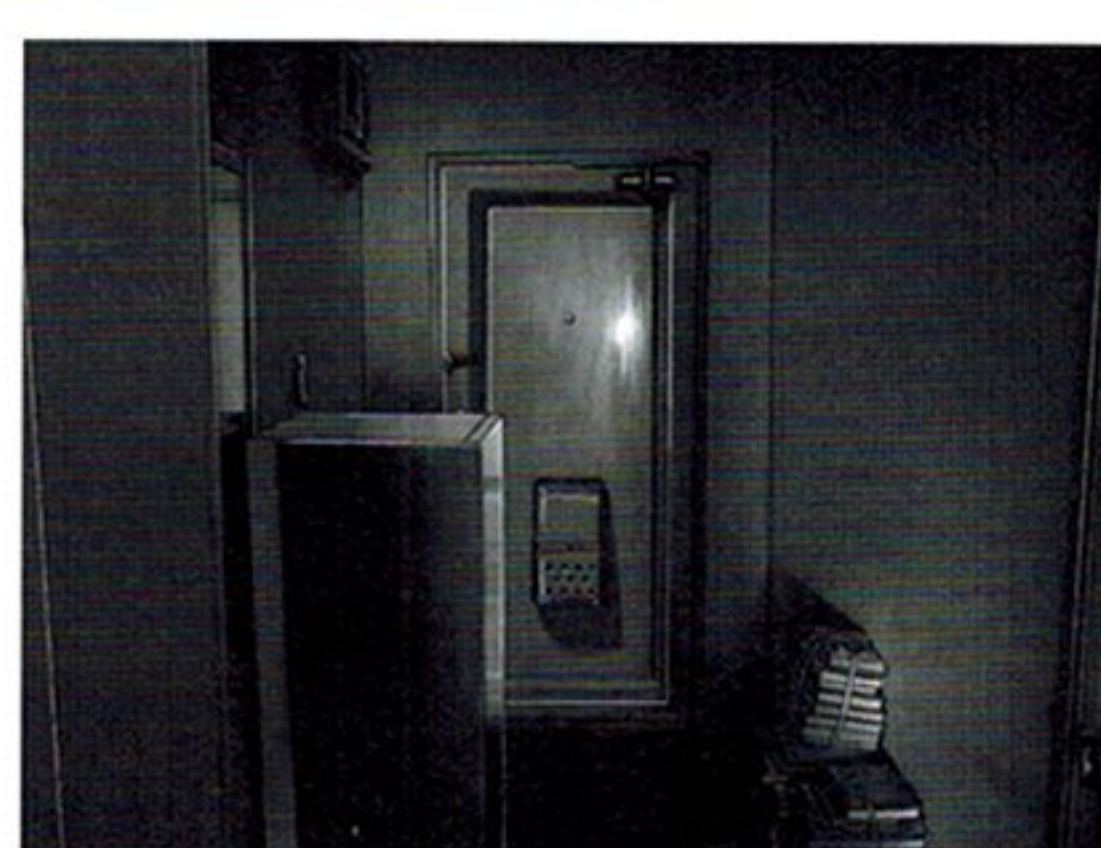



図50 例によって輪郭をトレースしてイラスト化。ベランダと同じく，テクスチャは写真をそのまま利用しているので作業時間は小1時間もあれば十分

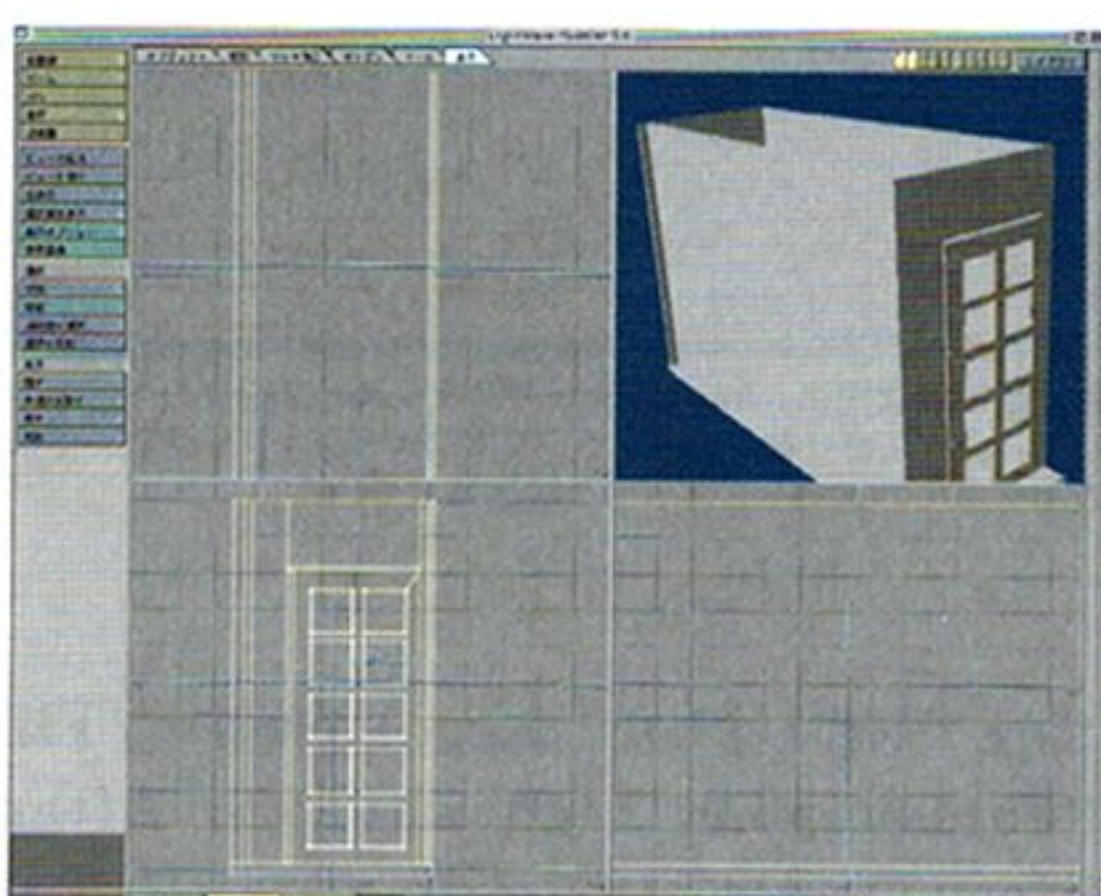



図51 図50のイラストにリビングのドアとその影のアニメーションを追加するために，室内を簡易モデリングする。リビングのドアはAfterEffectsでぼかしをかけるため単純なものでOK。また，室内の壁は単にリビングのドアの影を受けるためだけのものなので，Boxを組み合わせただけのもの

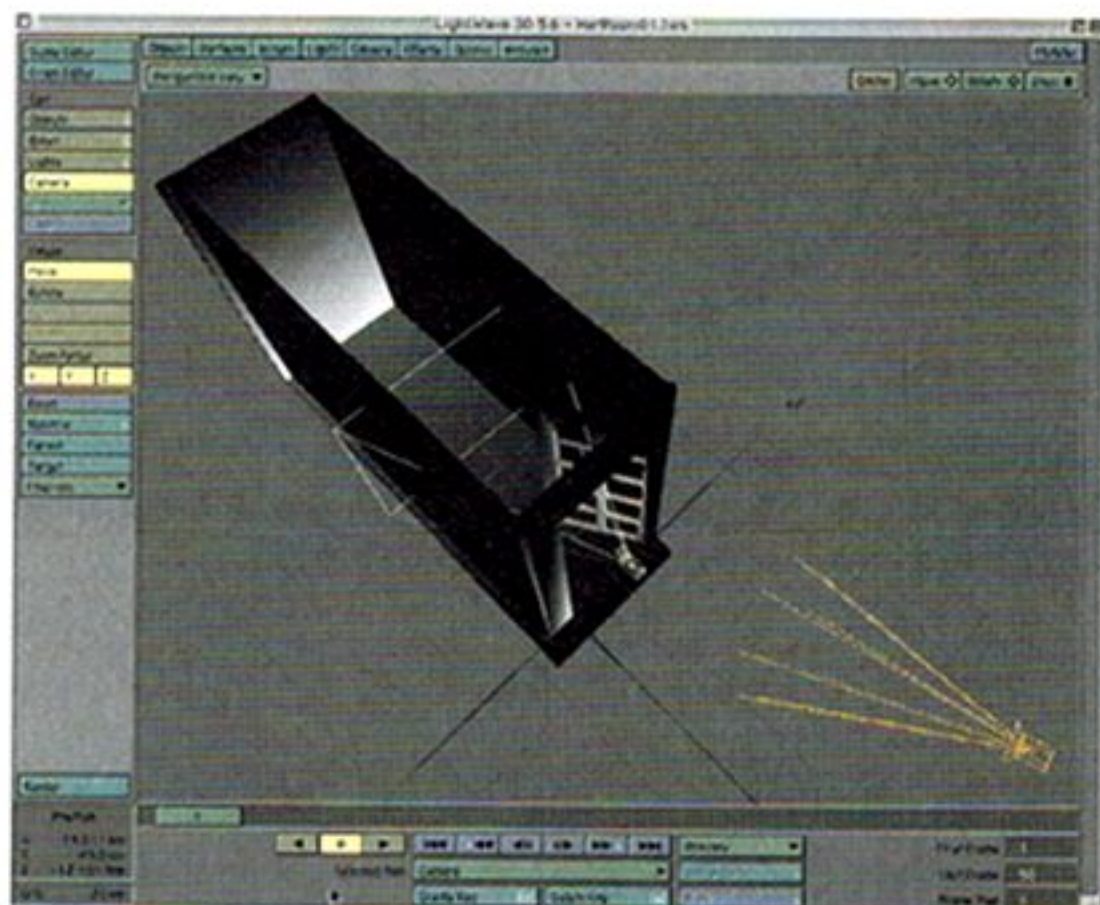



図52 LightwaveのLayoutで，室内のオブジェクトを読み込む。ドアのオブジェクトはピボットポイントを変更して，開閉ができるようにする。室内にドアの影を落としたいので，ライトは図のように配置する（オレンジ色がライト）

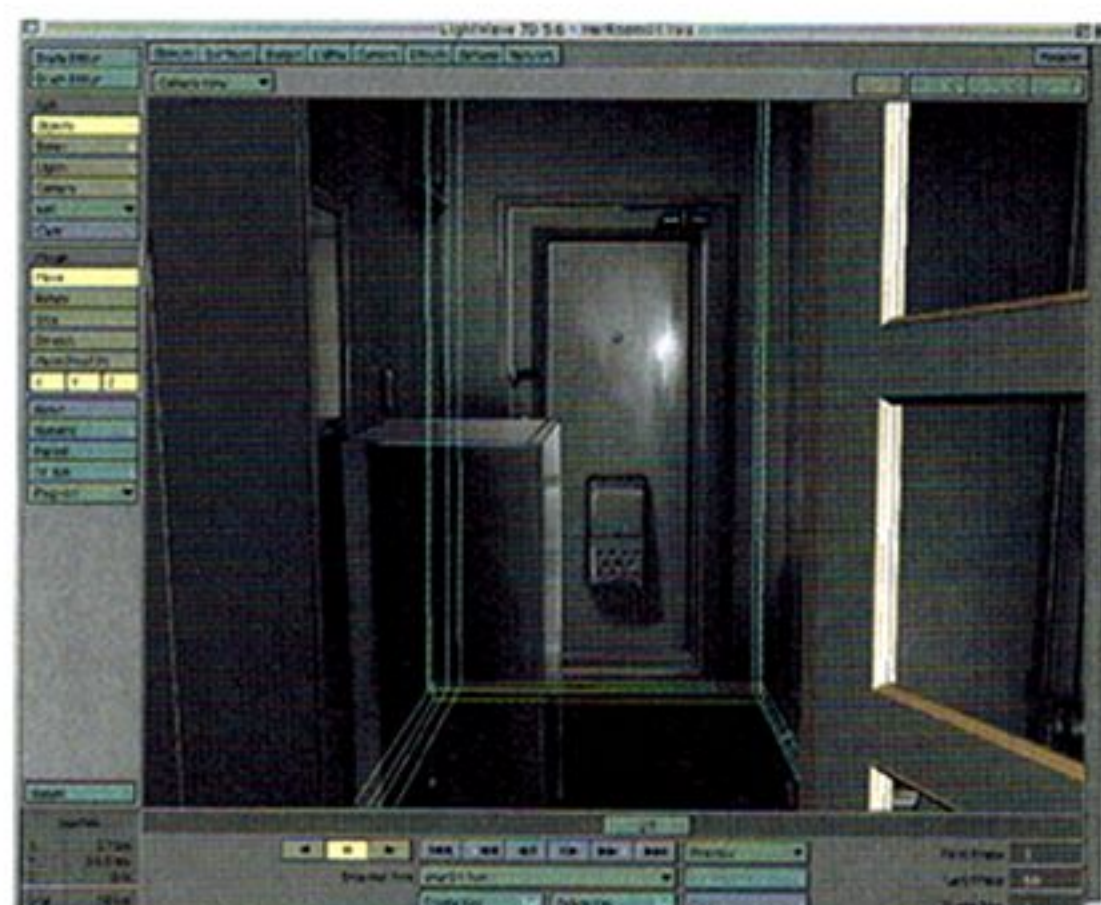



図53 Background Imageに，図50のイラストを読み込む。カメラを操作して，イラストと室内オブジェクトのおおよそのパースがあうように調整

図54 パースをあわせたらBackground Imageは削除し，テストレンダリング。ドアにキーフレームをつけてあるので，ドアの開閉によって後ろに落ちた影も動いている

を使用して作成しました（図49～59）。

涙を拭う手の動きの主観アニメーション

「だれか，だれか」というナレーションにあわせて，彼女の主観で涙を拭う手のアニメーションです（図60）。このシーンでは3Dソフトは使用していません。単純に動画を手描きしているのですが，キーフレーム的にデジカメ画像を使用しています。DVカメラなどをお持ちの方はそちらを利用したほうが作業はスムーズでしょう。また，できあがった動画には最終的にAfterEffectsでブラーなどをかけて仕上げています（図61～63）。

おわりに・制作のさらなる効率化を目指して

「完成形をイメージして，それを目指して仕上げていく」という過程は，オール手描きの2Dイラストでも3DCGでも，今回解説した写真をもとにする方法でもまったく変わりがありません。いうまでもありませんが，写真撮影にしても輪郭トレースにしても着彩にしても，もっとも重要なのは最終的に自分がなにを描きたいのかをはっきりと意識することです。描きたい内容によっては，写真をもとにするという手法が無効であることも少なくありません。完全にデフォルメ指向のキャラクターを描くには今回の手法は向きませんし，茂った木の葉や草原などの自然物も，複雑すぎてトレースには向きません。反対に比較的リアル志向の人物や背景の作画には，今回解説した方法は効率化のための非常に有効な手段となるでしょう。

3DCGの華やかなアニメーションが花盛りの昨今ですが，我々が長年親しんできた2Dタッチのイラストにもやはり抗いがたい魅力があるのも事実で，そのような方向を志向する方に今回の記事

図55 テストレンダリングでうまくいけば，ドアとその影を別々にレンダリングする。これはドアのみのレンダリング画像。ドアのアニメーションはアルファチャンネルで抜きたいので，背景の室内オブジェクトをObject Dissolve100%にして一時的に見えなくしている。イラストと馴染ませるためセルシェーディングでレンダリング

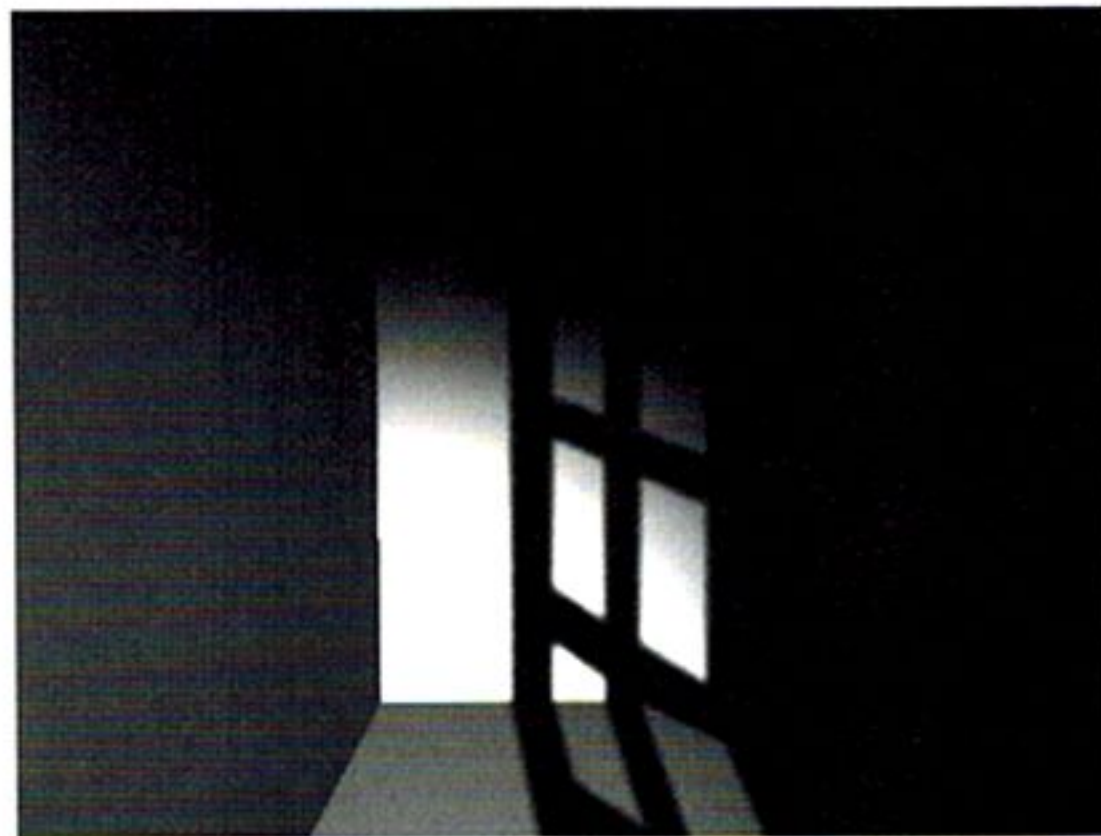



図56 こちらは影のみのレンダリング。ドアが邪魔だからといってObject Dissolveで隠してしまうと，肝心のドアの影が落ちなくなってしまう。そこで，スキャンラインの影には透明度は影響しないことを利用して，ドアのサーフェスの透明度を100%にしてレンダリングしている

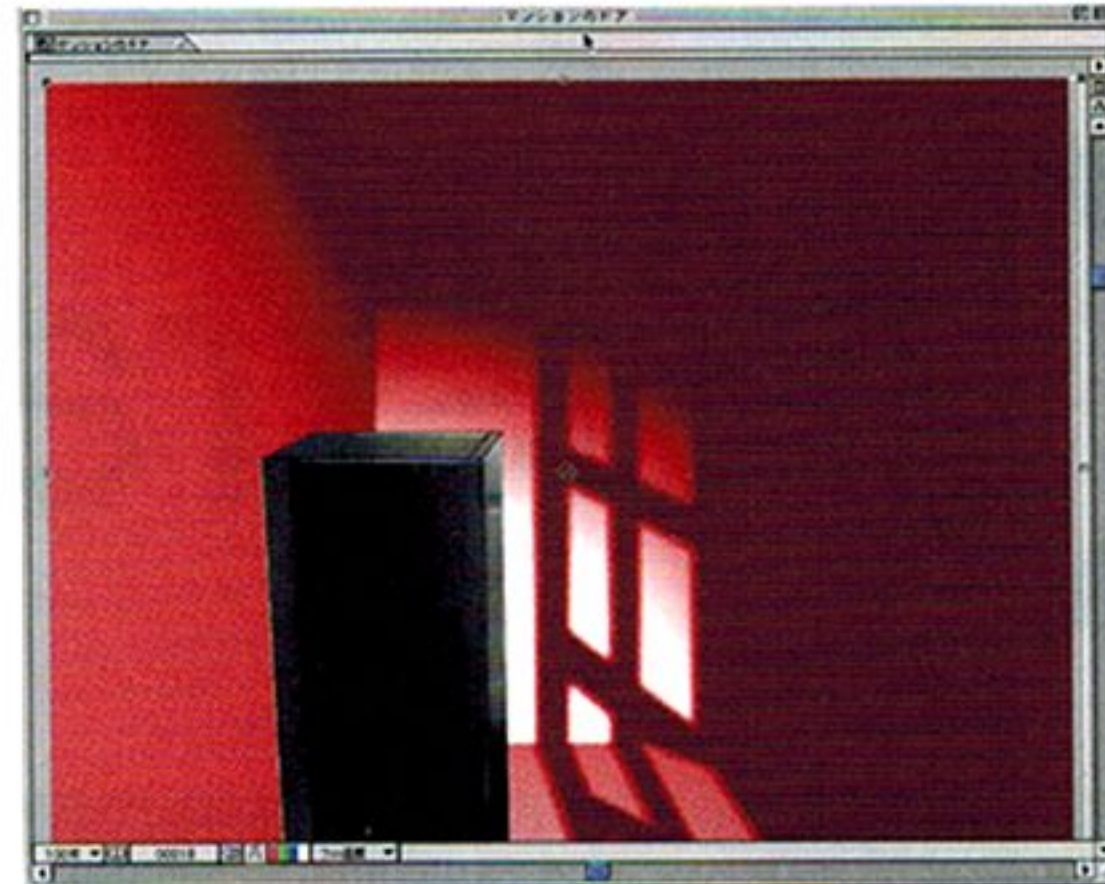



図57 図56の影に必要なマスキング処理を行う

図58 AfterEffectsで図57のマスクを使用して，イラストに図56の影アニメーションを重ねる。重ね合わせモードはソフトライト，透明度は80%。パースの違和感もなくイラストに溶け込んでいる

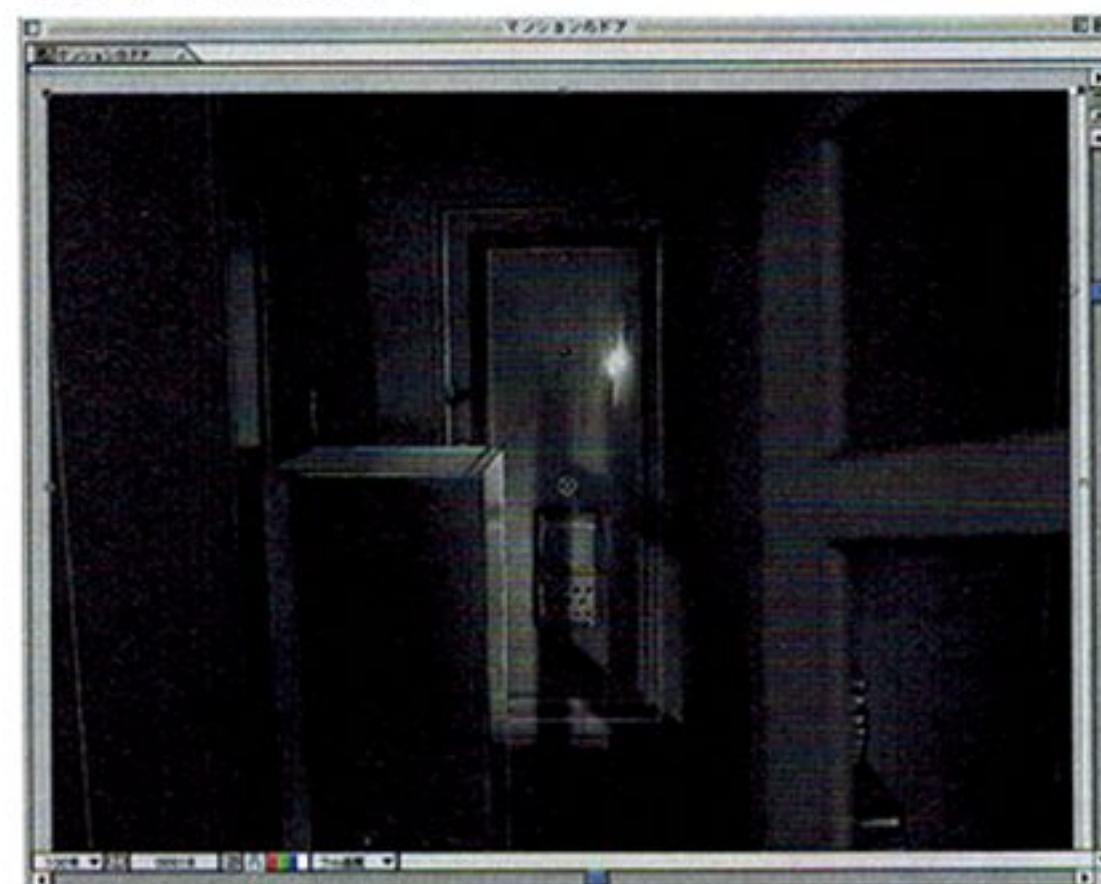



図59 さらに図55のドア開閉アニメーションを重ねる（ここではわかりやすくするため，重ねたドアを茶色にしている）。カメラの焦点は奥のドアにあっているということにして，重ねた手前のドアにはブラーをかけてイラストに馴染ませ，完成

図60 完成ムービー

図62 Photoshopで描いた1枚1枚のレイヤーをAfterEffectsに配置していく（上のタイムレイアウト）。当時はAfter Effects3.1を使用していたのでレイヤーの配置は手作業だったが，4.1からはキーフレーム補助のレイヤーシーケンス機能で自動配列できる。さらに，そのようにしてレイヤーを配置したコンポジションに，動きに応じたブラーや色調補正を手動で行う（下のタイムレイアウト）。右の図面がブラーをかけたもの。手が手前に近づくほどブラーが強くなるようにしている

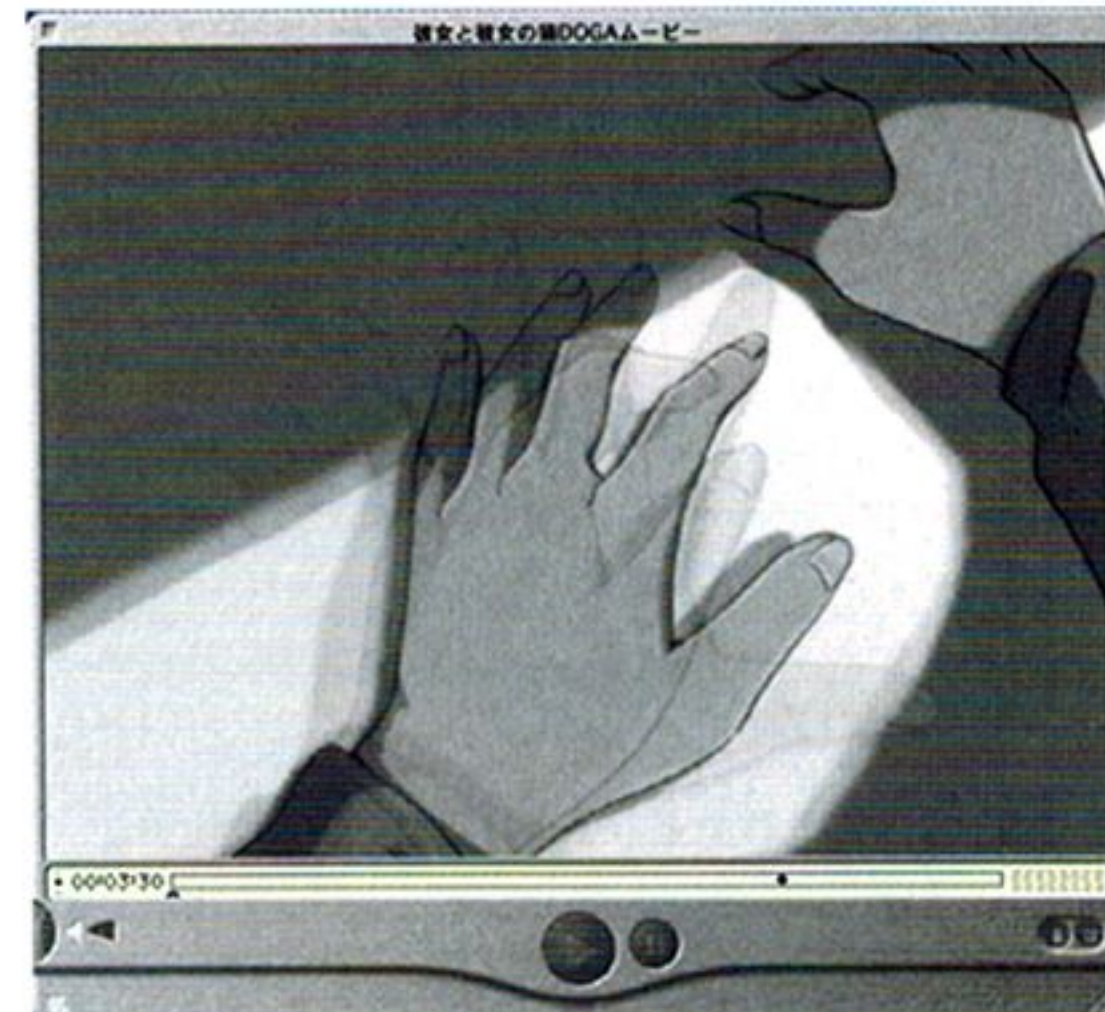
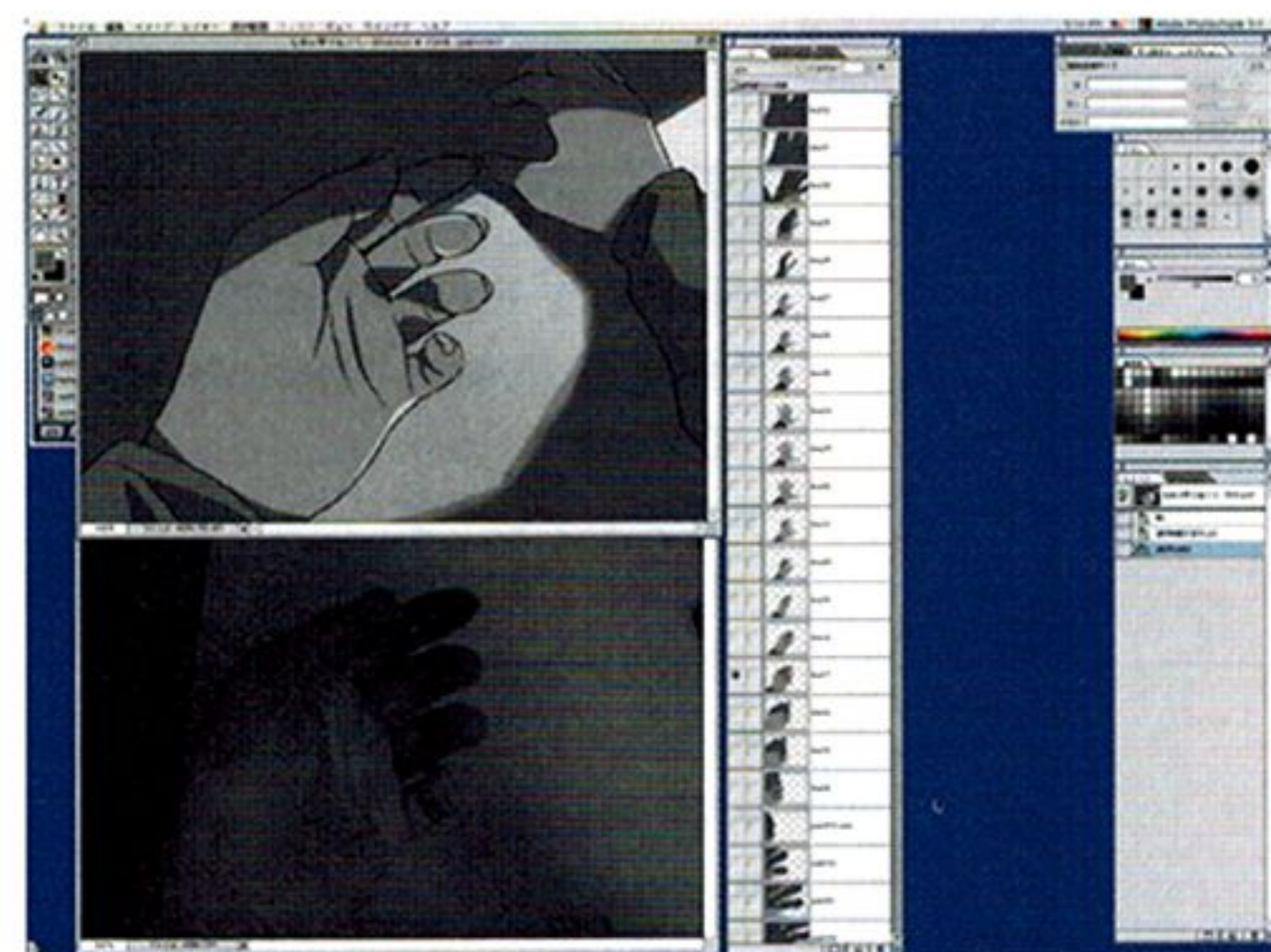
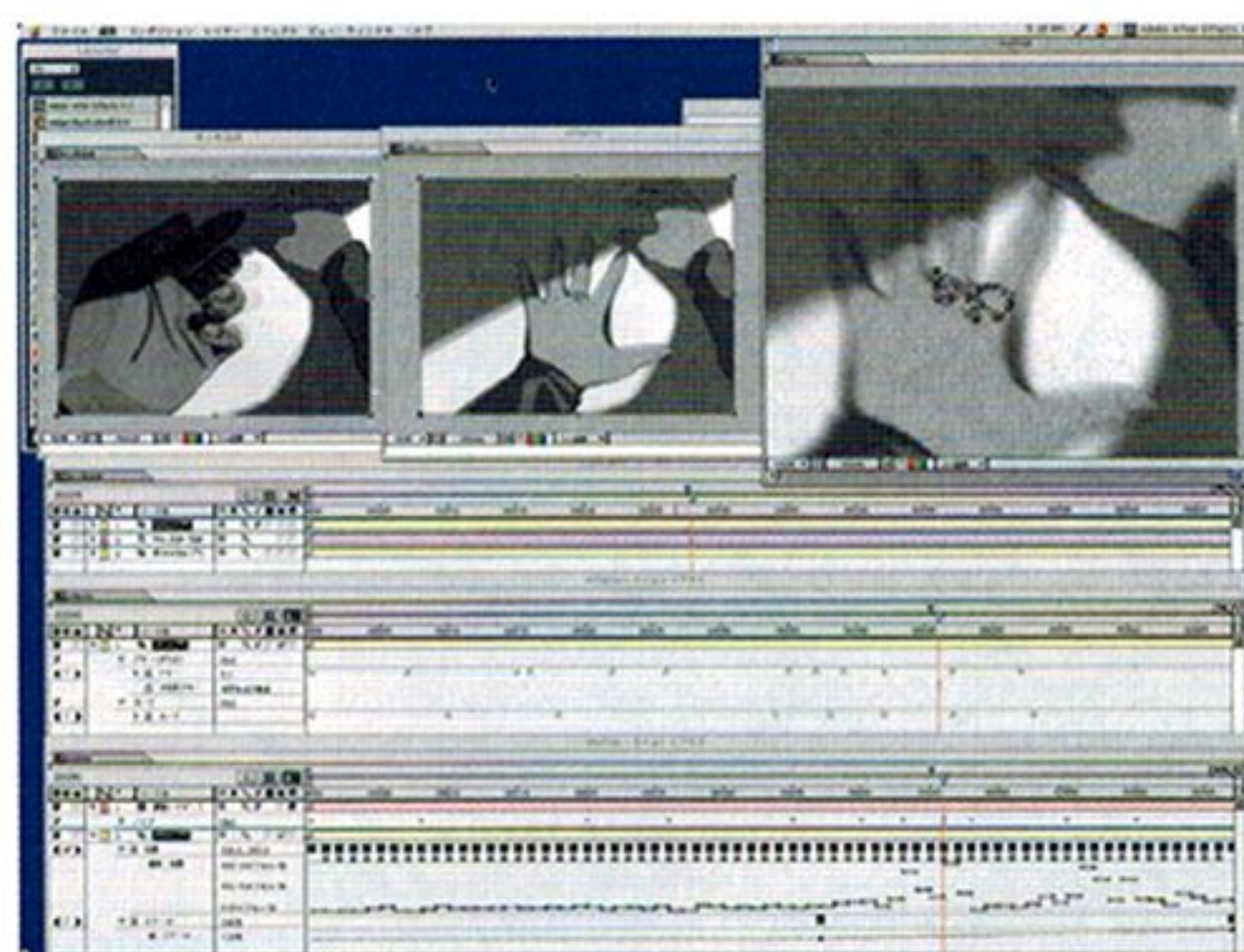



図61 デジカメでキーとなる手の動きを数点撮り，トレースしてイラスト化する。キーフレームの間を埋めるかたちで，その他の動画も描いていく。この手の動きは全部で32枚の動画を描いている。レイヤーパレットをご覧いただければわかるが，動く部分（左手）のみを背景とは別のレイヤーに描いている。これは背景と左手で個別にブラーなどをかけたいため

図63 図62で作成した左手のアニメーションを背景と合成（左の図面）。背景も，手の動きに応じて影がアニメーションするように作ってある。さらに，その合成アニメーションそのものにもブラーとトーンカーブのキーをつける（中央の図面）。涙で視界が滲むような効果と，手が近づいたときに視界が暗くなる効果を狙った。最後に仕上げとして，いままでのコンポジションよりひと周り小さなサイズのコンポジションにこれまでのアニメーションを配置して，位置とスケールにキーをつけている。これは視界（カメラ）のブレを表現することで，より切迫した感じを出したかったため

がなんらかの参考になれば幸いと思います。

私事になりますが，『彼女と彼女の猫』がナレーションと効果音と静止画主体というスタイル（しかもモノトーン）をとっているのは単純に制作時間の確保が重大問題だったからで，「もっと絵を動かしたい」という欲求はもちろんありました。たとえば今回解説したような手の動きのアニメーションなども，デジカメ画像を補助的に利用することでアニメータではない私にもそこそこ現実的な期間で作成できていますが（実質2日ほど），それでももの足りない感はどうしても残ってしまいます。

ここはぜひ効率的なワークフローを確立しなければと思い立ちまして，先日デジタルビデオカメラを購入しました。FireWire（i.LINK）経由で動画像をダイレクトに取り込めるDVカメラは，手描き調2Dアニメーションを志向する者（かつ非アニメータ）にとって，作業のさらなる効率化を可能にしてくれます。そのあたりの手法も，またの機会に解説させていただきたいと思っています。

Illustratorで主線を描く

Youi Kawahara 川原由唯

マシン環境

相変わらずPowerMac7500です。

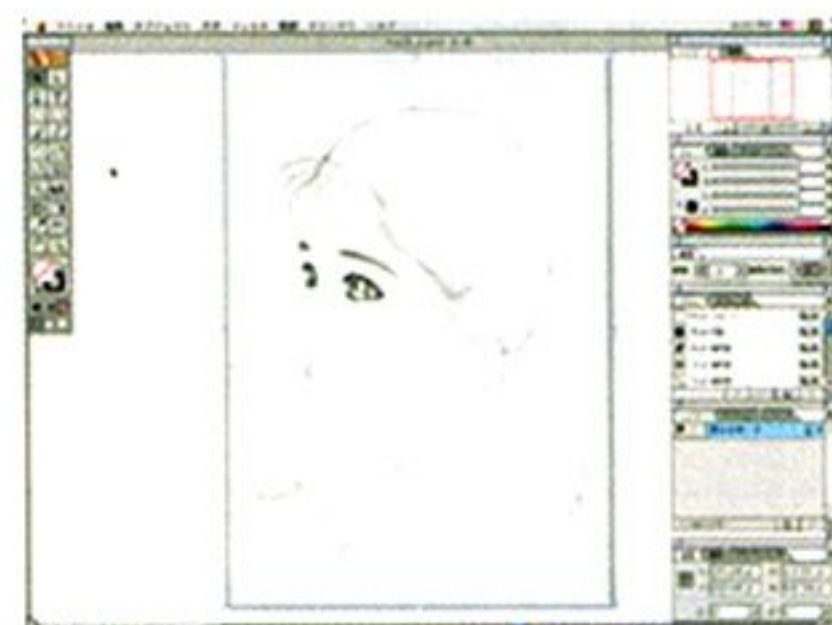
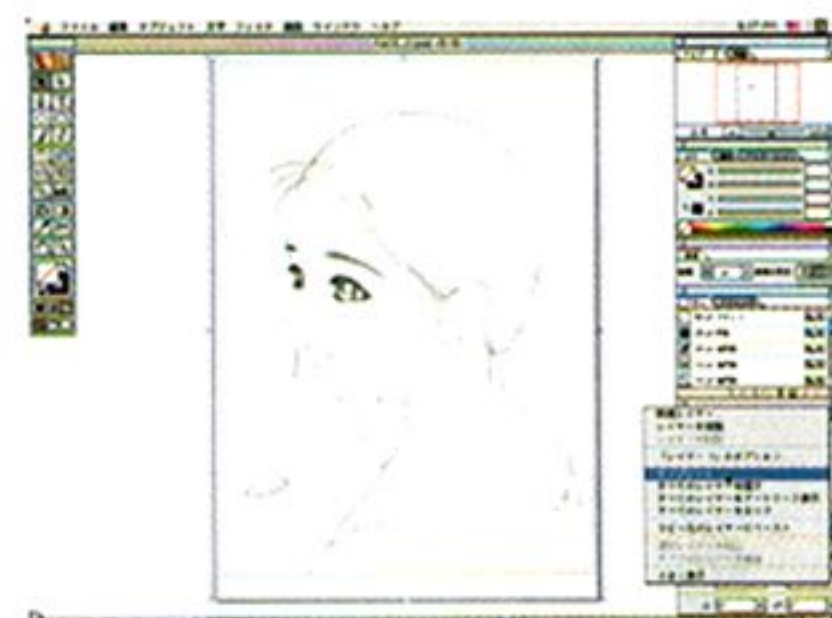
- Apple Power Macintosh 7500/100
- アクセラレータ XLR8 MACh SPEED 604e 233MHz(250Mで使用)
- 搭載メモリ　352MB
- VRAM　4MBに増設
- ハードディスク 内蔵1GB＋4GB＋外付け9GB×2のソフトウェアRAID
- ディスプレイ 17 inch TRINITRON
- タブレット　WACOM UD-608
- スキャナ　JX-250
- 230MBのMO，CD-R，フィルムスキャナなど
- MacOS 8.5
- Adobe Photoshop 4.0.1J (5は重いからいまだに4！)
- Adobe Illustrator 8.0

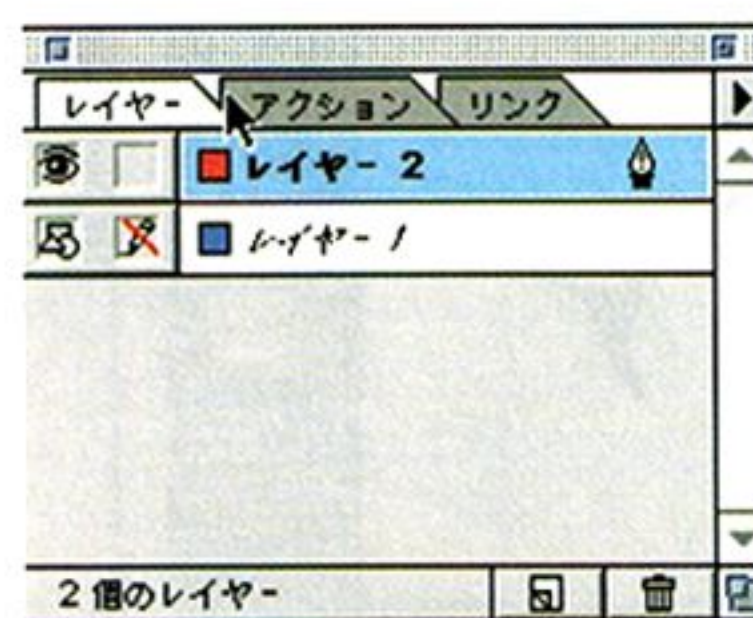
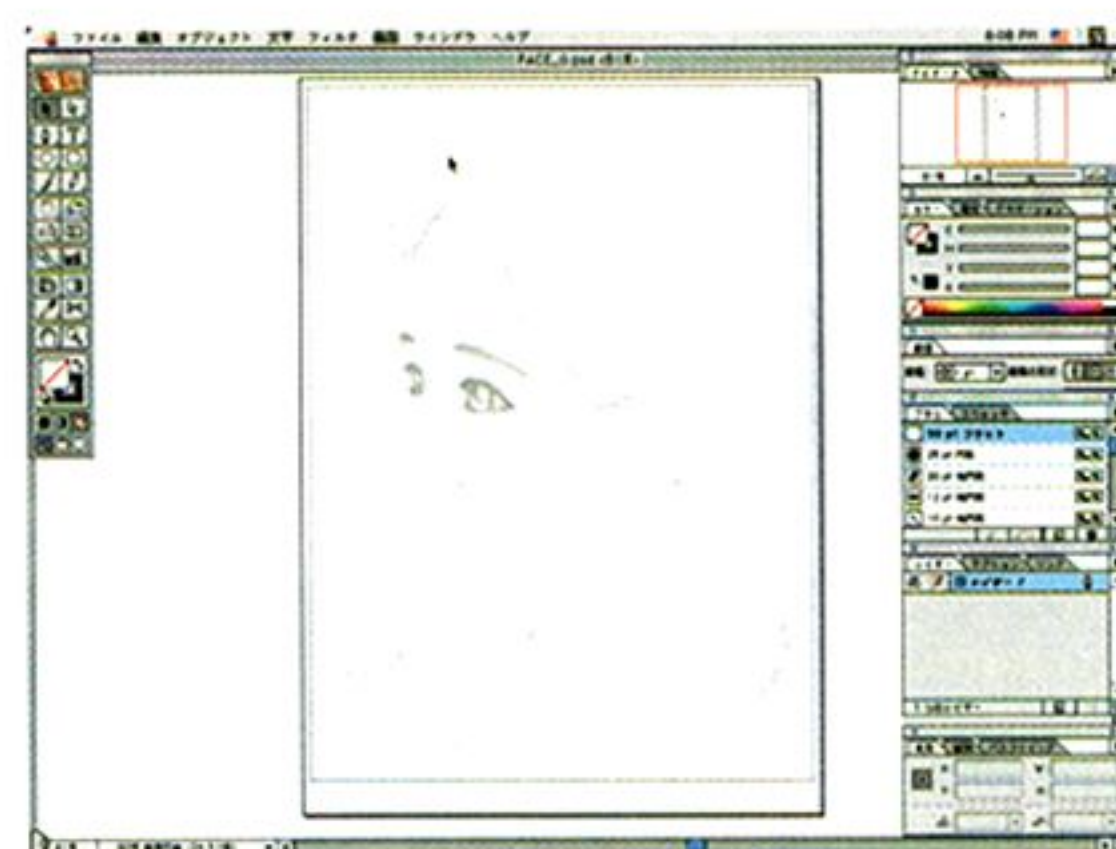
MacOS8.5にしたら，Photoshopが妙に遅くなったような気がするのですが……。早くG4買いたいです。といいつつ最近はLinux用ATXマシン(組み立て)に財産注ぎ込みすぎて資金流出。あうぅ～。

ドロー系ツールの練習を兼ねて，"Illustrator 8.0" を使ってみました。 Photoshop と同じ Adobe 社の製品だけあって両ツールの親和性はバッチリで， Photoshop に慣れたユーザーなら，パスやアンカー操作などのドローツールの本質的なこと以外はほとんど悩むことなくシームレスに双方を行き来できると思います。ライバルの "Macromedia FreeHand" に比較して若干機能が劣るといわれても，僕が使う程度の機能なら必要十分でしょうし，Photoshopとの連携の安心感は大きなアドバンテージになっています(両方使ったわけではないので偉そうにはいえませんが……)。

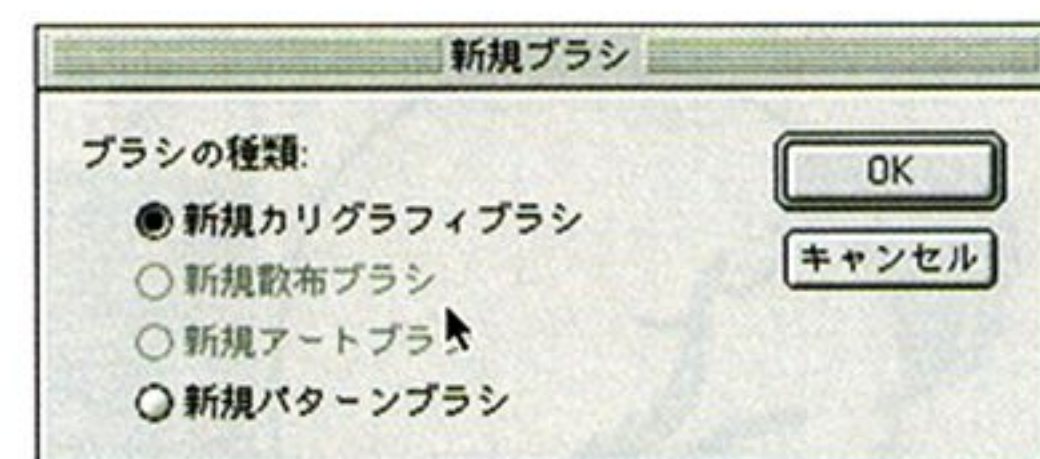
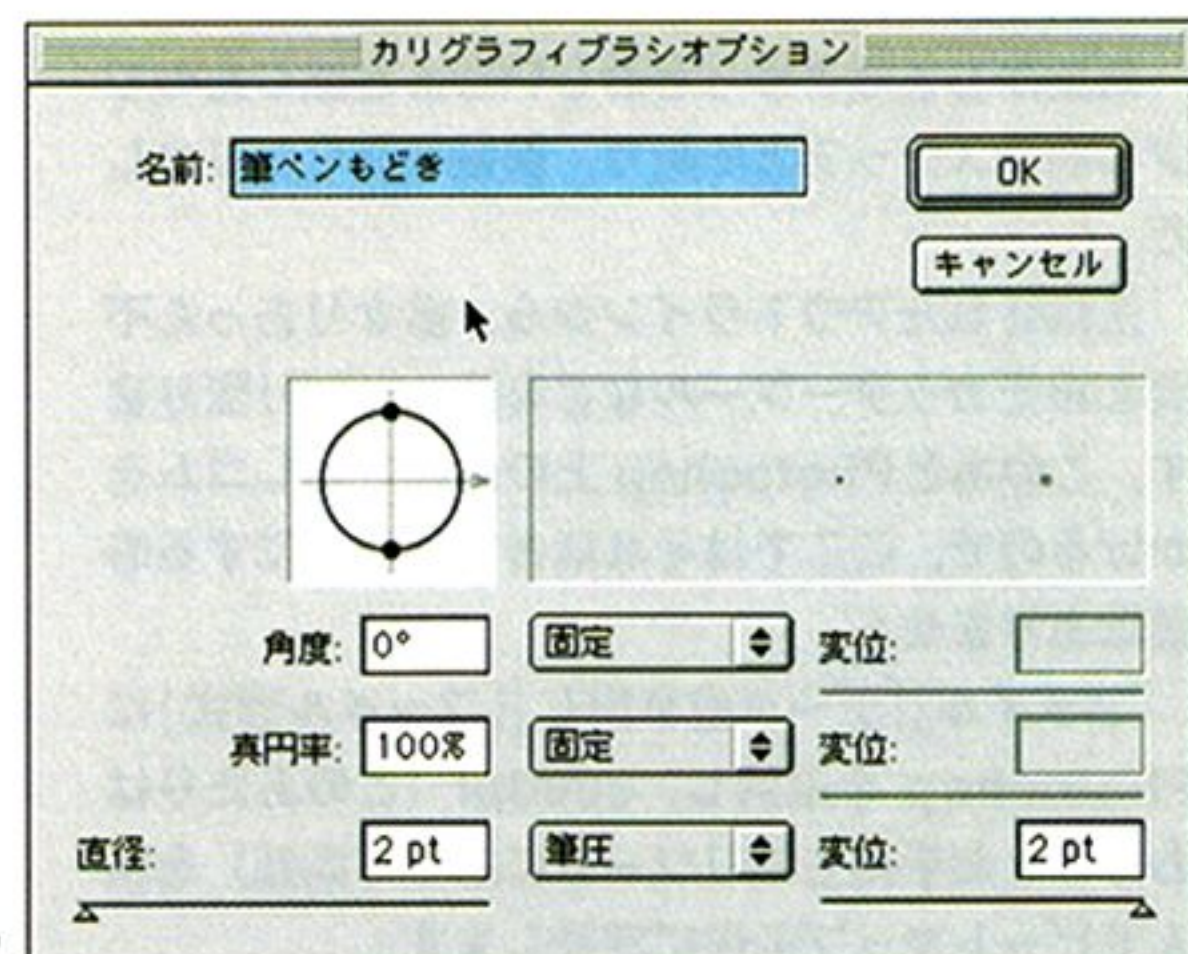
さて，毎度恒例お約束のいい訳をひとつ。僕はドロー系ツールに関しては超初心者 (UNIX 上の tgif をときどき使っていた程度) ですので，「これから使い込んでやるぞ～」という意気込みはあれど，ごくごく初歩的ことしかわかりません。ここに書いたことももっとも基本的な使い方のひとつと思われます。それでもこの記事を見て，ペイント系ツールしか使ったことのない絵描き予備軍の皆さんにドローツールという存在を知ってもらうことができれば (意外と知らない人が多いんだよね) 書く意味もあるのではないかと思います。というわけであえて駄作をドウゾ。

下描き

毎度ワンパターンですが，KMK-KENT紙にシャープペンで描いた下描きをスキャナで読み込みます。いつもの僕の絵だと，取り込んだ線画を最終的な作品の主線にしてしまう場合が多いんですが，今回はあくまで主線を引くためのアタリでしかないので，かなりおおざっぱに描いてあります。PSD (Photoshop) 形式で保存しておき，Illustratorで読み込みます。

下描きを取りこんだところ

テンプレート

Illustratorには下描きの画像データを置く専用のレイヤーである「テンプレート」があります。レイヤーのメニューで「テンプレート化」すると，取り込んだ PSD の絵が薄く表示され，トレースするための下描きとなります。

テンプレート化するときのレイヤーメニューはこんな状態

テンプレートレイヤは淡く表示される

ペンの調整

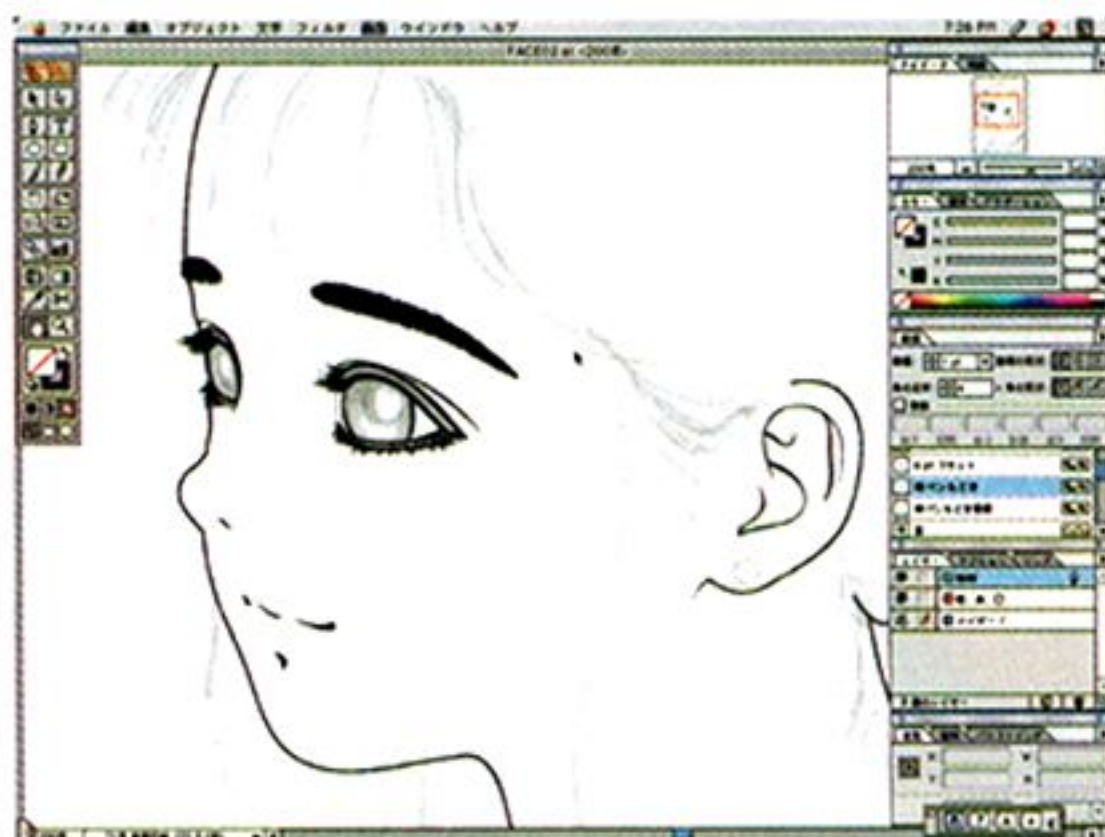
Illustratorの描線であるカリグラフィブラシは感圧タブレットの筆圧に応じて線の太さを変えることができます。デフォルトで提供されているペンは太すぎるので，好みで微調整して，新規カリグラフィブラシとして登録しておきます。

カリグラフィブラシを作る

パラメータのウインドウ

ペン入れ

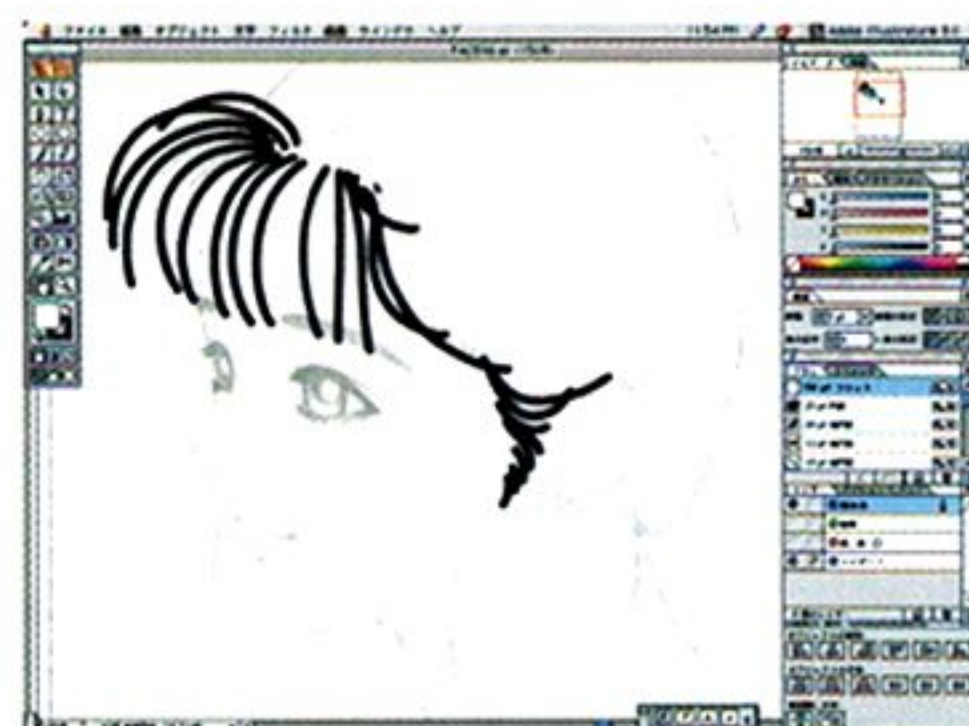
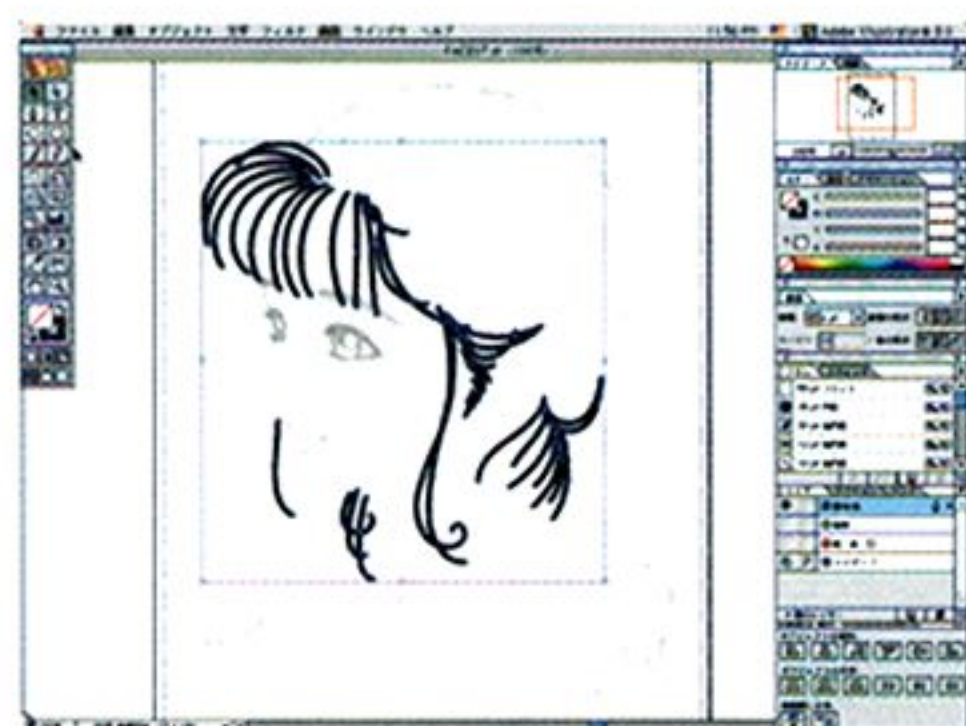
このブラシの使い方には少しコツがあります。本物のペンと紙を使って描画するときは，ペンを抜く方向に向かって線が細くなりますが，Illustrator のブラシはペンを入れる方向を細く (つまり筆の運びを逆向きに) したほうが描きやすいようです。

顔の輪郭線は余計なタッチをつけたくなかったので，均一な太さの線を描ける「鉛筆」ツールを使って線引きします。手が震えて線が多少ゆがんでも，あとから1本ずつ削除したり，微調整できるのがドローツールの利点。少々のブレは気にしないでザクザク描画します。

Photoshopでも使われる手ですが，各パーツ，たとえば瞳，眉，髪の毛，輪郭線ごとに別レイヤーを作成し，分離させておくとあとあと編集がやりやすくなります。

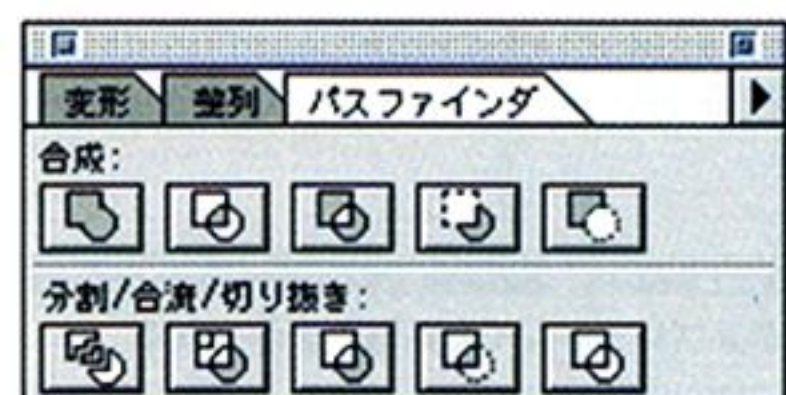
筆圧を加減しながら描くとこうなる。まつげの先から付根に向かってブラシを走らせるのがポイント

どんどん描く。ひたすら描く。眉毛の毛だって1本1本描けちゃう

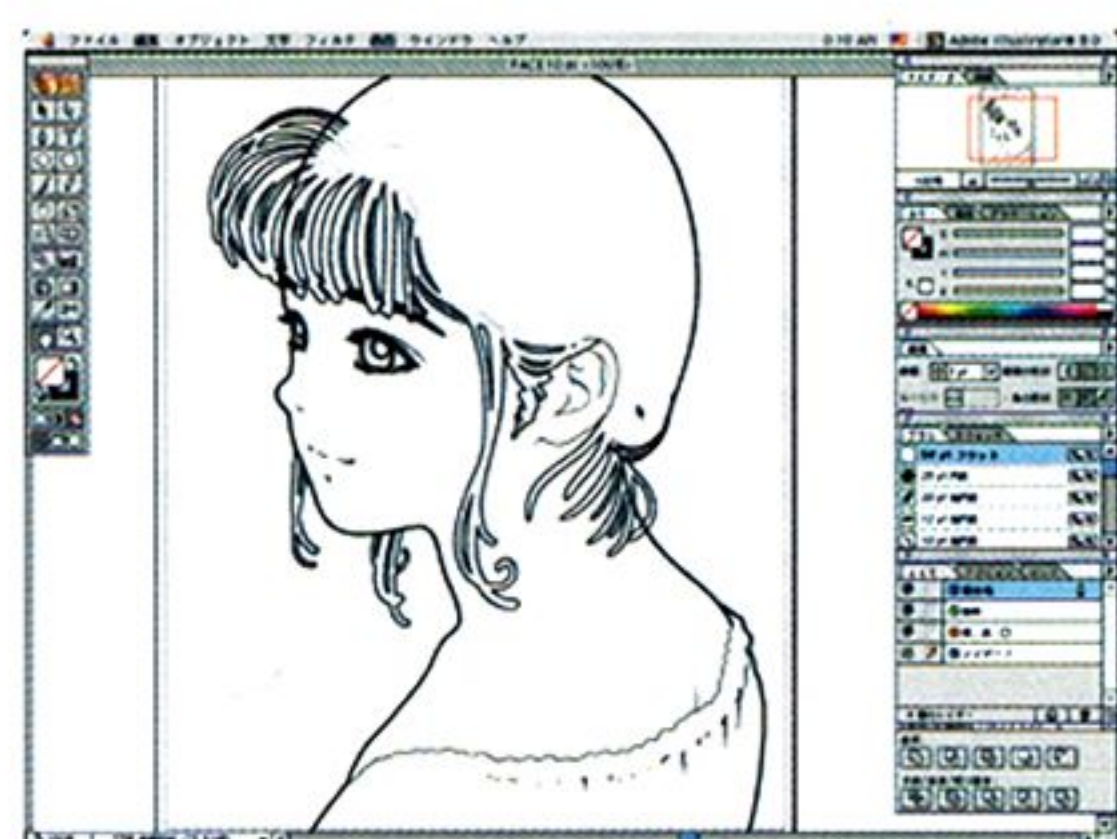
先に作成したカリグラフィブラシは使わず，太めに設定した鉛筆ツールで描きます。

この髪の毛の描線については，まだ研究の余地があると思っています。このあとの作業で「アウトライン抽出」を実行し輪郭線として使うために，カリグラフィブラシで描くことができないのですが，鉛筆ツールを使って描くと太さがいかにもコンピュータで描きました的な均一になってしまうために絵としての面白さがなくなってしまうんです。このあたりは個人的に今後の課題にしています。

髪の毛を描く

ひたすら描く

太さを適当にバラすと少し自然になる

線の太さを調節するパレット

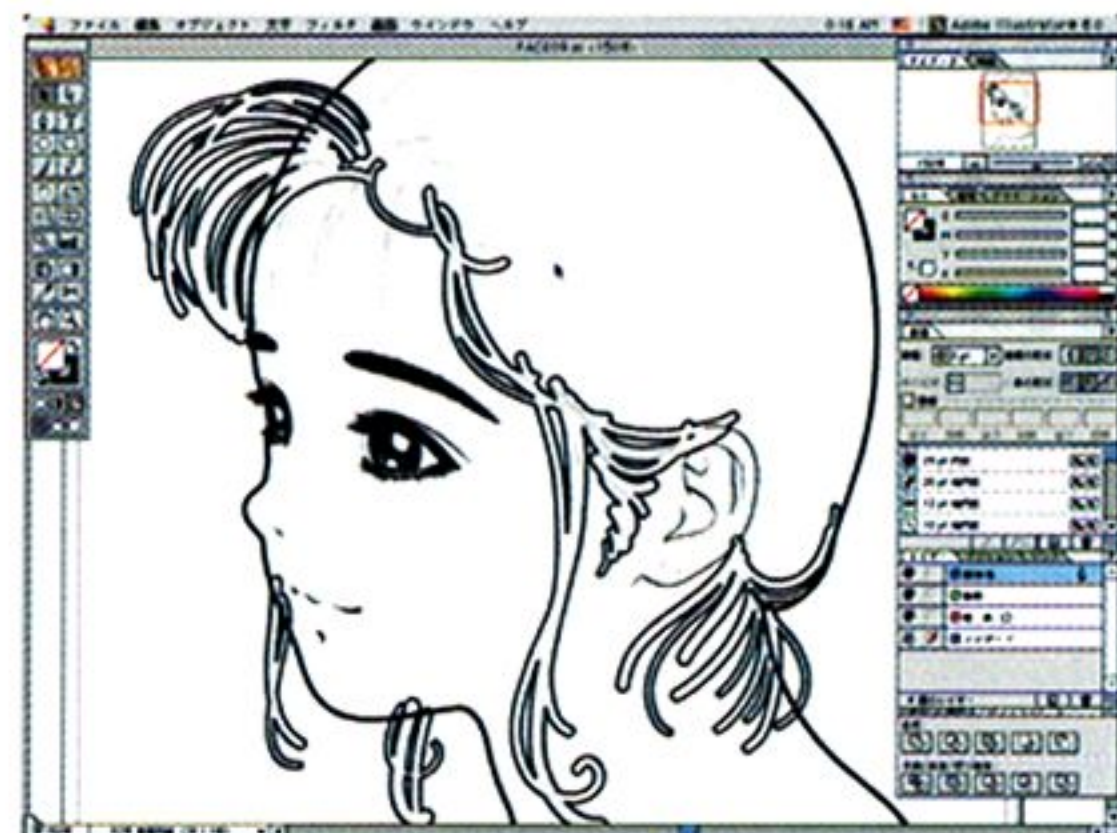
髪の毛のパーツだけ選択し，パスファインダのパレットで「合体」をしたうえで「オブジェクト」「パス」「パスのアウトライン」を抽出。こうするとミュシャの女性画みたいっしょ？

線の太さを調節するパレット

輪郭だけ抽出するとこんな感じ

パーツ類を全部表示してみたところ

前髪がちょっとうっとおしいなぁと感じたので少し前のステップまで戻り，前髪を削りとりました。

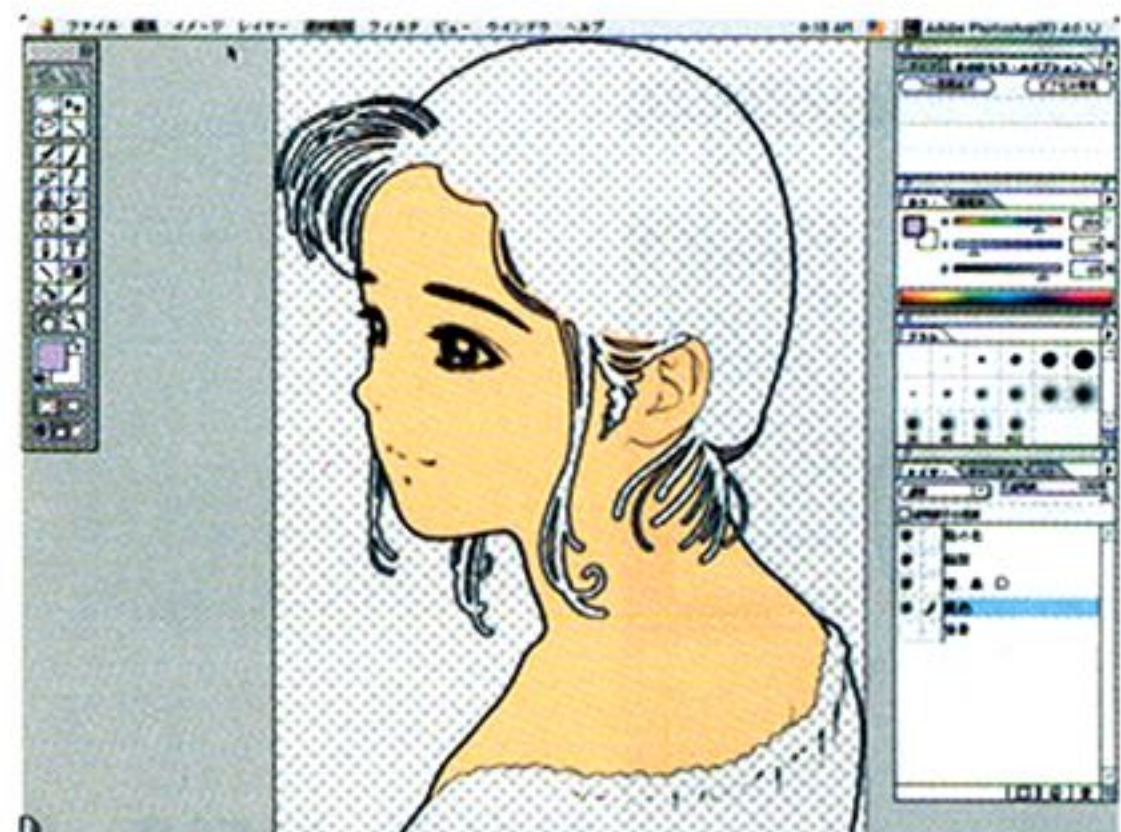
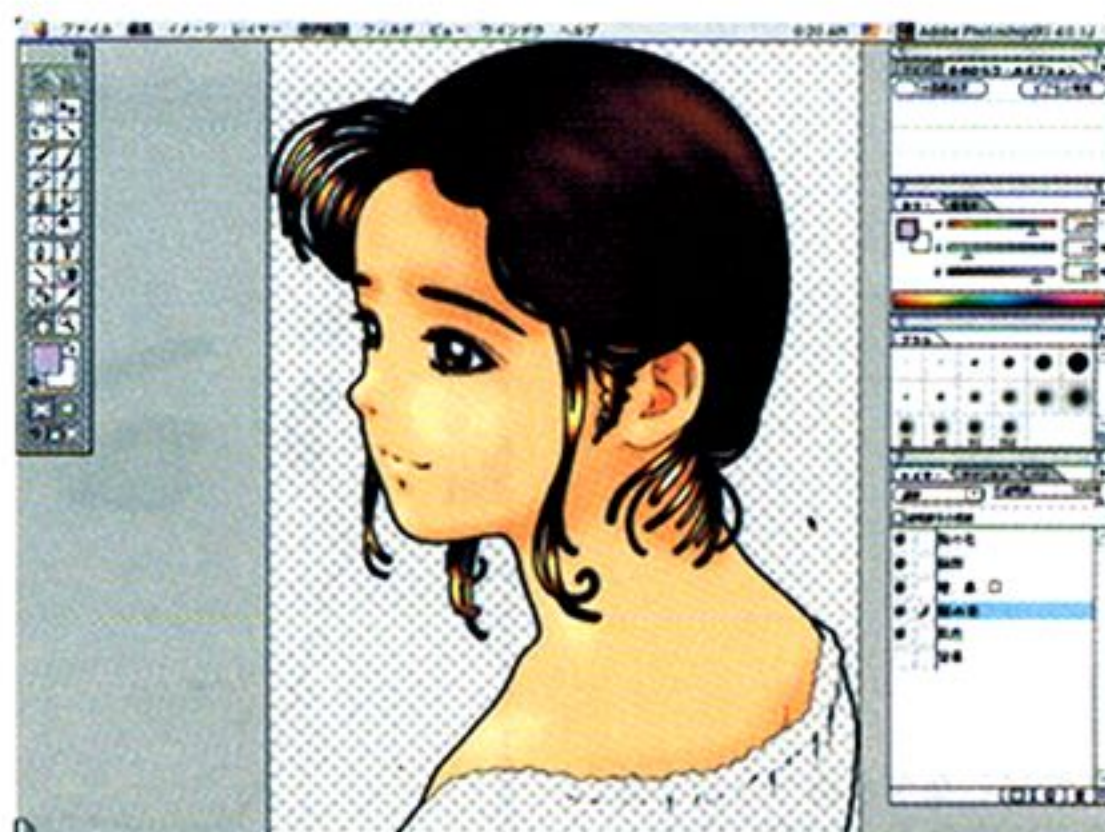
生成されたアウトラインから，重なりあった不要な線をカッターツールなどを使って削り取ります。このあとPhotoshop上の処理で消しゴムをかけるので，ここではそれほどていねいにする必要はありません。

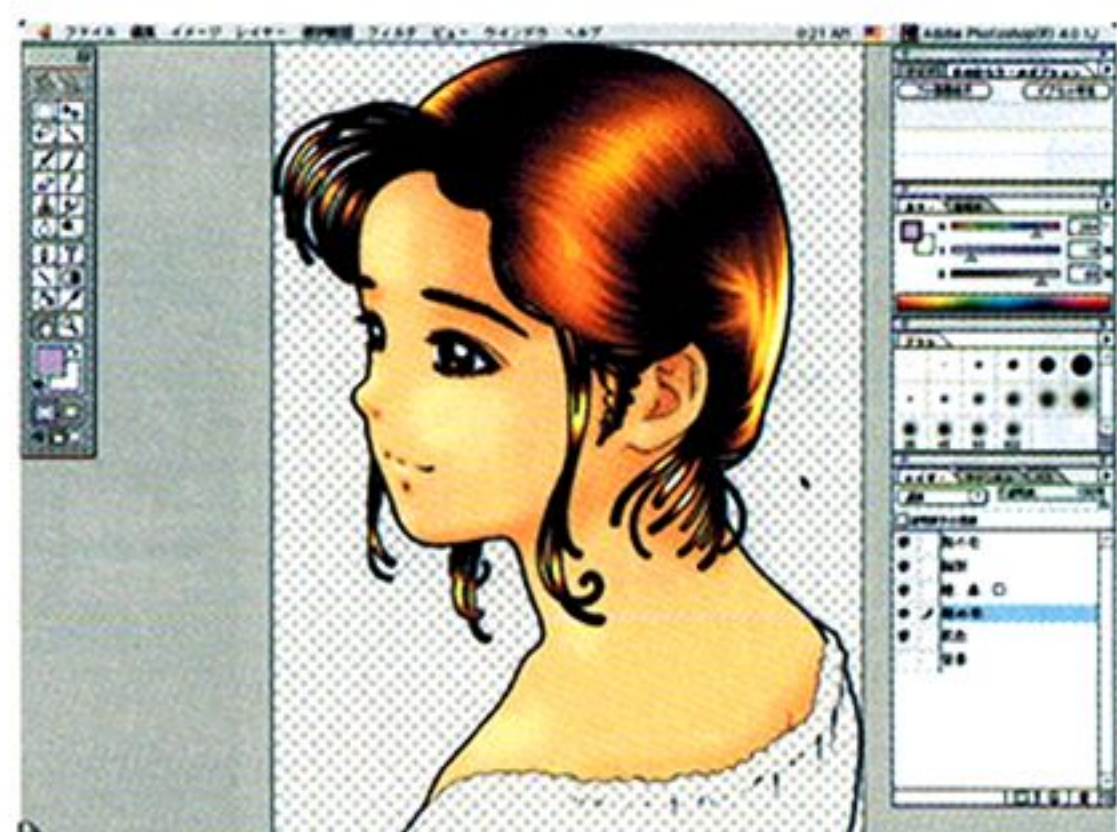
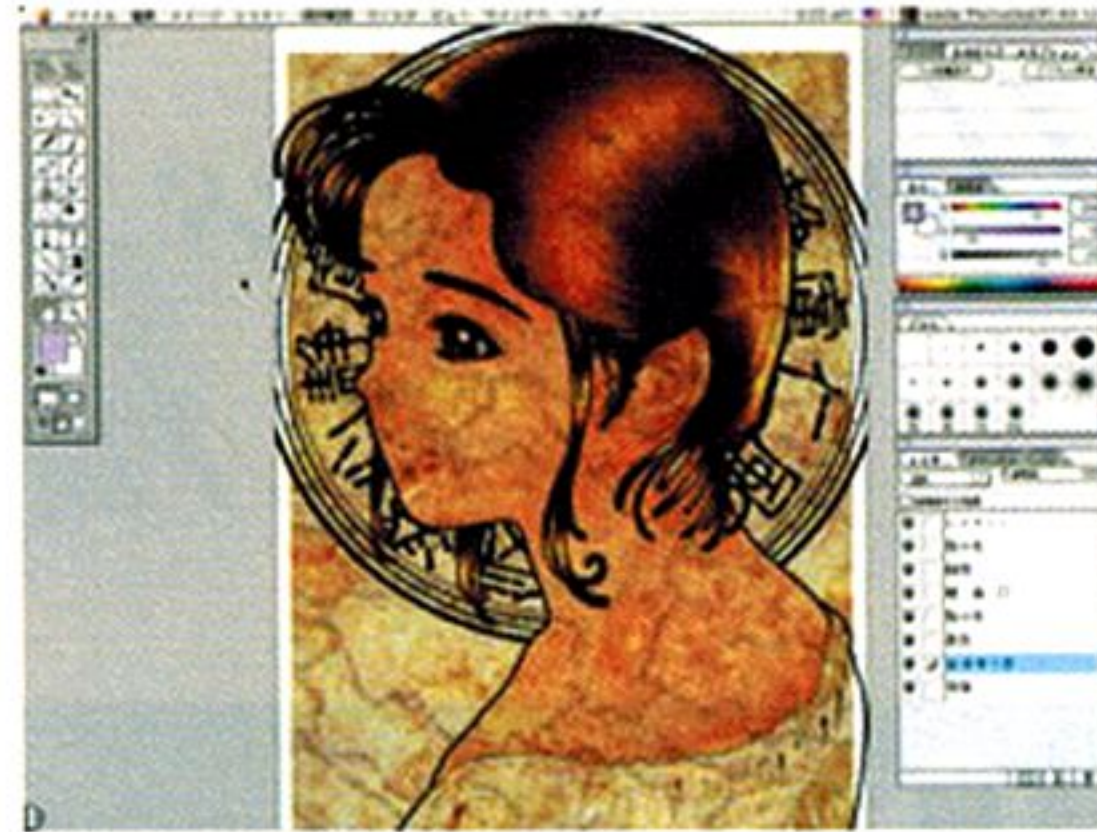
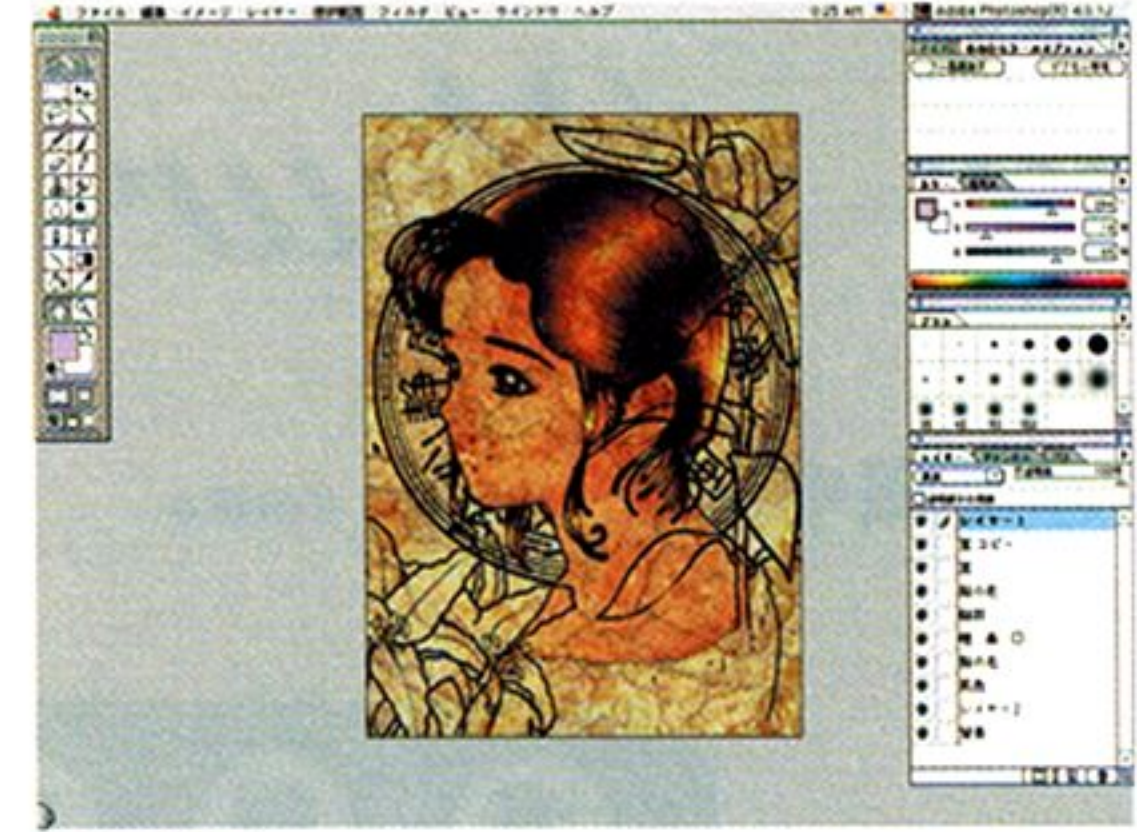
「ファイル」「データ書き出し」「ファイル形式」にPhotoshop を選択し，300dpi（このあたりはお好みとお手持ちのリソースに応じて調整）を選んでビットマップ画像に変換します。

前髪の線を削って調整したところ

ふたたび髪の毛のアウトライン抽出

Photoshopにて編集

ここからPhotoshopの作業に移ります。もちろんIllustrator上でベタ塗りくらいはやっておいてもいいのですが，各パーツがクローズパスになっていなかったりすると思いどおりの着色ができないので，この絵でのIllustratorの作業は線画だけに割りきっています。

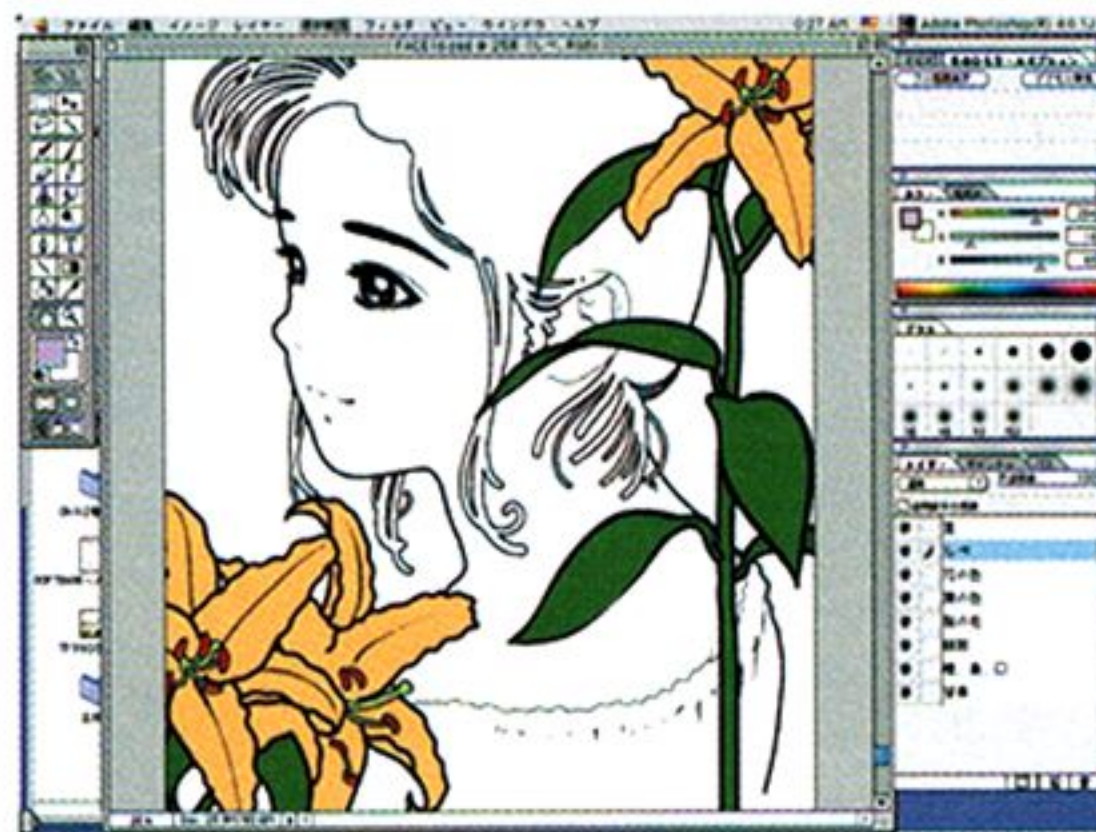
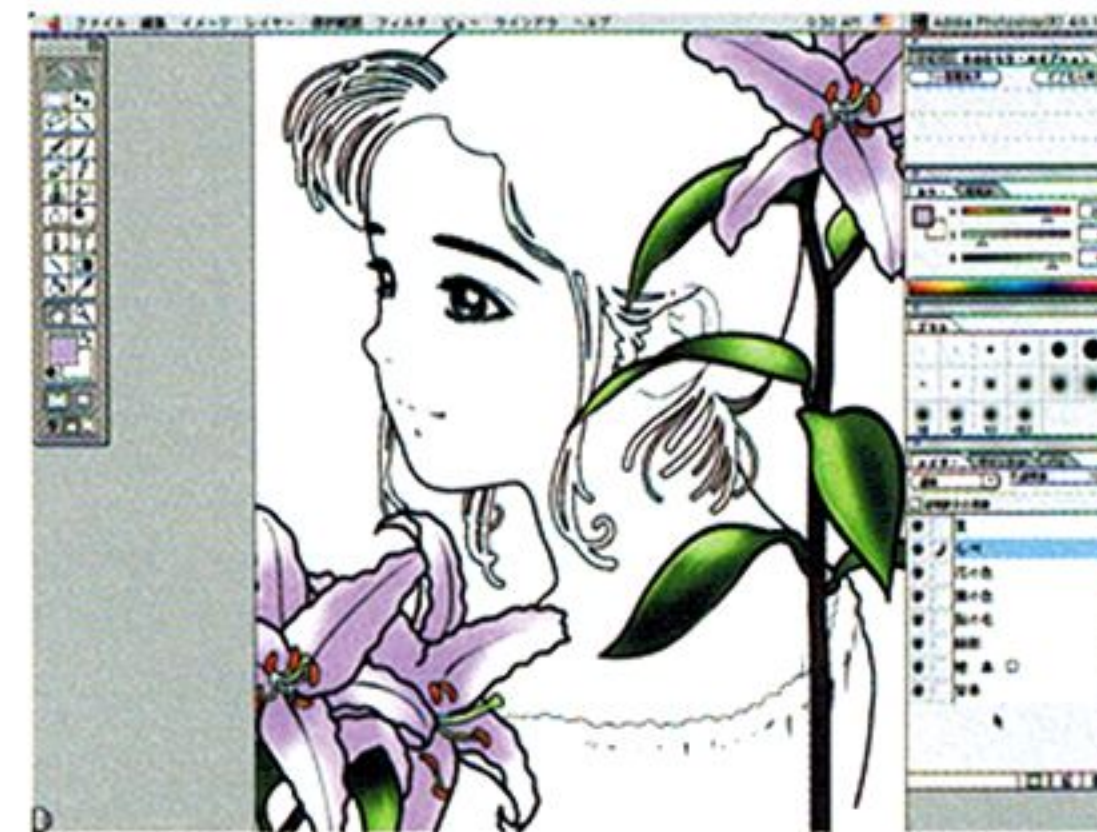
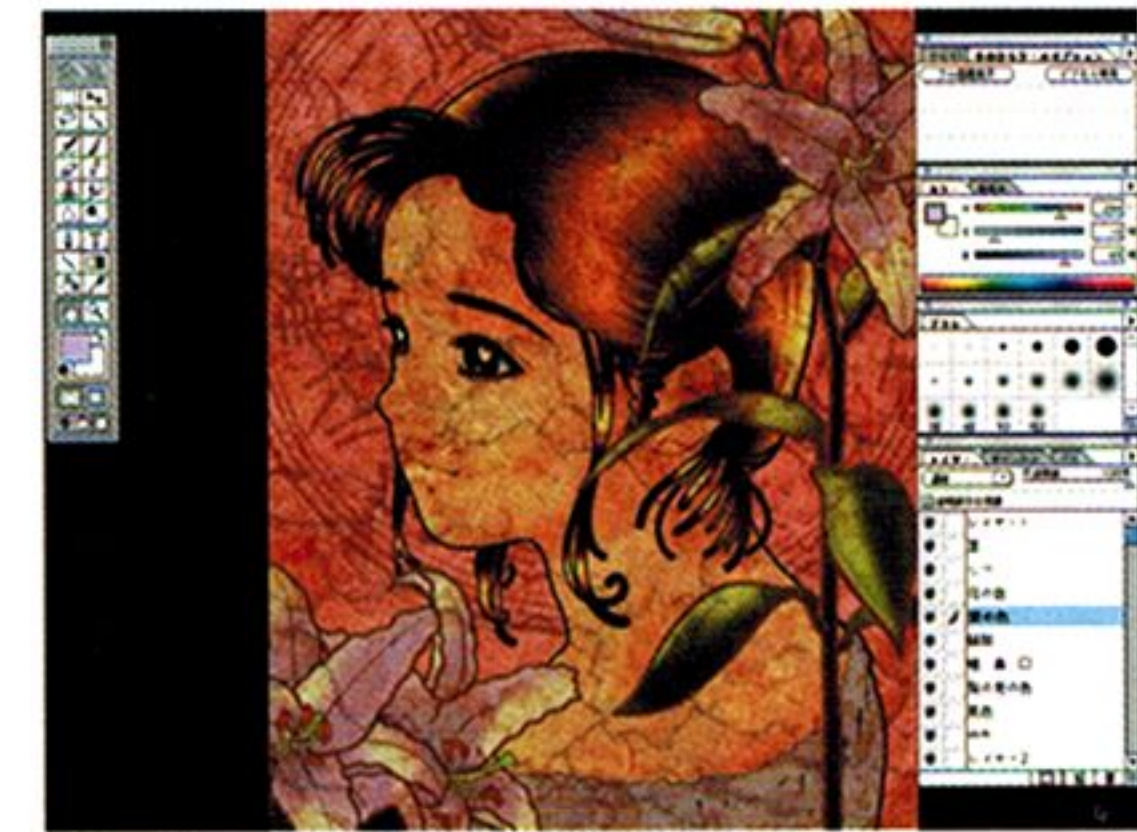
書き出したデータをPhotoshopで読み込みます。読み込んだだけですでに線画（セル画）状態になっているはずなので，すぐに各種ブラシツールを用いて着色できます。

Photoshopで読み込んで塗り始めたところ

特に説明することなし。ひたすら塗ったり描いたりこすったり

これも（おぃおぃ）

背景埋め用の画像

合成してみる

位置の調整中

時計の文字版，もしくは魔方陣のようなデザインをIllustratorで作成し，背景に置いてみます。こういったデータを作成するのがIllustratorの本領発揮の用途。ってゆーか，僕もこういう絵を描きたいがためにIllustratorがほしくなったのです。いやぁ，CCさくらさまさまだよね～（え？）

古びた雰囲気を醸し出すために，素材集の大理石を上に重ねてみました。なかなかいい感じ。これで進めてみることにします。

さらにミュシャっぽくするために（?）ユリの絵を重ねます。このユリも鉛筆で描いた下描きをIllustratorでなぞって作成したもの。

人物と同様に色を乗せていきます。
あとで色の調整することを前提にしているので，細かい色調は気にしないで陰影だけに注目して描く。
最後にパーツの位置や透明度などを調整しつつ、完成。今日のところはこのくらいにしといたろか。

花の着色中

簡単に陰影をつける

最後にパーツの位置や透明度などを調整しつつ

できあがり

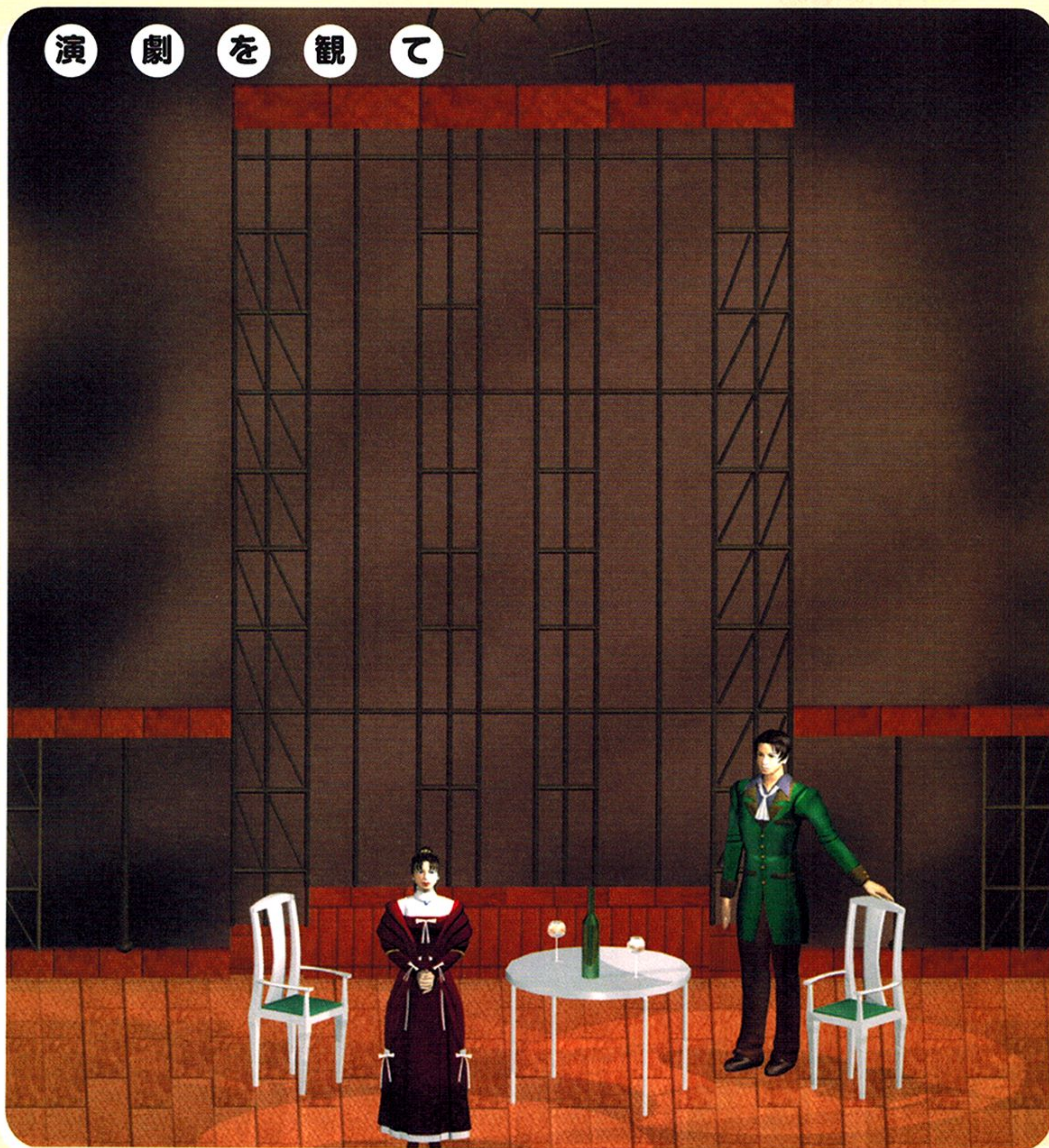
最後に

まだパスの編集に慣れていなかったり，思ったとおりのタッチが出せていなかったりで，全然使いこなしている満足感はありません。もっと勉強＆練習しなければなりませんね～。でも普段描かない（描けない），太く綺麗な主線が簡単に描けるのはなかなか楽しいものです。また瞳やまつげのあたりは予想以上に綺麗に描け，自分の画風にも意外とあい，満足のいく仕上がりになることがわかって使い甲斐がありました。

ここでは紹介しませんでしたが，Illustratorは選択領域への文字列の流し込みなど，文字配置の機能も豊富なので，簡単な組版ならPhotoshopと組み合わせて十分活用できそうです。いままでPhotoshopだけで苦労してレイアウトしていた「ちらし」のようなものも，とても簡単に作ることができました。このあたりの用途にはもっと早く使ってみるんだったなぁって感じ。もっともこの手のソフトはプロユースのツールだけあって高価で，気軽におすすめできないのは残念なところです。

Your Memories, My Memories

田中順子 Tanaka Yoriko

演劇を観て

よいお芝居というのはとても不思議です。舞台という小さな世界で，舞台の背景や，俳優たちの情熱ある台詞を見聞きしているうちに，人間の欲や夢，そこにある風景などが，私たちの胸に現実として現れてきます。まるで，自分がその世界の住人のようにお芝居というイメージの世界を体験してしまうのです。

特に俳優たちは，私たちをそのイメージの世界に案内する素晴らしい役目を果たしています。俳優たちの歌には登場人物の切ない気持ちや胸に秘めている複雑な感情が強く表現され，情景がよりいっそう深まり伝わってきます。また，彼らが踊れば，舞台は明るく美しく華やかになります。そして彼らの語る言葉やしぐさは普通の会話であっても私たちの胸に強い印象となって入ってきます。私たちはそんな彼らに感動し，涙を流すのです。

私は年に数回は必ず劇場に行くと決めています。それは，落ち込んだときなどいい気分転換になるというのもあるのですが，仕事上こういった芸術文化に触れるのは，私にとてもいい影響を与えてくれるのです。それに，お芝居という世界に陶酔したあと，パンフレットを眺めながらさっき観た物語を友人とお茶を飲みながら熱く語るというのも結構楽しいものです。

REAL VIRTUAL GAL'S WORLD

ファンタジーゲームと服の関連性

野沢 絵美　Nozawa Emi

ファンタジーというジャンルが一般に受け入れられるようになってから，ファンタジーの小説，マンガや，ゲームといったものがワチャーと市場に登場しました。10年近くファンタジー作品を追っかけている私としては，昔と比べてファンタジー作品の選択の幅が増えたのでうれしくなっちゃいます。

なぜファンタジーが好きなのかと問われると，ヨーロッパちっくな舞台設定が素敵とか，登場人物の王子様とか若い魔法使いが艱難辛苦の末に難問を解決するって感じのストーリー展開がたまらないと答えるのですが，もうひとつファンタジーを追いかける楽しみがあります。それは服装です。豪華絢爛な服装とか日常ではお目にかかれないデザインの服装にストーリー中やイラストで出あえるのがうれしいのです。たとえば，高級なシルクと高価な宝石をふんだんに使った王族の衣装，金と銀を編み込んだサンダル，頭からつま先までをすっぽり覆う魔法使いのマント，ドラゴンや怪しげな文様が彫刻されている鎧などなど。ゲームパッケージなんかに大胆にデザインされた素敵な服装のイラストが登場しているとフラフラ～と手に取ってしまうこともしばしばです。

ファンタジーというジャンルは自由度が広いがゆえに，敵と戦い経験値を上げるようなゲームでは頻繁にファンタジーの設定が使われます。ところが，ファンタジーというのは自由度が高いがゆえに，世界観をしっかり設定しないとせっかく作ったゲームの世界も現実味のない薄っぺらな作品になってしまう危険性もあります。世界観をしっかり作り，ある程度の決まりや制限をその世界に持たせることで，ファンタジーの世界は現実味を帯び，ユーザーがのめり込みやすくなります。実際に素晴らしいと思える小説やゲームなどは見事なほどに世界観が決まってます。ファンタジーの世界観を設定するには，世界観，時代，国，気候，交通手段，宗教といったものがあります。今回はそういった設定が服装に反映されること，反対に服装にも表れます……といったお話です。

ファンタジーの世界設定で必要な項目

それでは実際に架空のファンタジーの世界を設定しながら説明していきましょう。ファンタジーの世界を作るときに必要な設定項目は，だいたい下記のものとなります。そのほかにも作りたい世界によっては必要な項目が出てくる場合がありますが，一例として参照してください。

・時代設定（技術）
・時代設定（通信/交通網）
・国と人種の違い
・宗教
・地形と気候の設定
・地域別産物
・文様，文字，模様

世界設定がはっきりしていると，服装も自然とその世界に似合ったものが浮かんでくると思います。なぜならば，人が身にまとうものはいつでもその時代と土地風土を反映しているからです。現代のように各国の状況をテレビで見れる時代なので，日本とかけ離れた気候を持つ舞台でもある程度容易に服装の想像がつくと思います。

それでは簡単な世界を設定しながらこれらの項目の内容を説明してみましょう。

●仮想設定（大まかなストーリー）

中世のヨーロッパに近い世界で，日常生活に飽き飽きしていた少年が夢に見た女の子を探し求めて旅立つ。しかし，それは世界を救う冒険の始まりでもあった……。という単純ストーリー。

●世界設定（ヒーローの場合）

・時代設定（技術）

剣で戦う中世時代あたりがモデルだが，独自のエネルギー源とそれを利用した技術があるので，自動車，船に似た乗り物が存在する。ただし飛行手段はない。

・時代設定（通信/交通網）

海路，陸路は発達。

通信手段は手紙や人伝えによるウワサ程度。

・彼が住んでいる国と人種の違い

海路による交通が盛んであり，商人や交易により稼いでいる。多くの人種が入ってくる。

・宗教

キリスト教に近い宗教が存在。街の人間は信仰はそれほど深くないが，船乗りを職業とする人間は信仰心が強い。

・地形と気候の設定
温暖な地中海性気候。フランスのニースあたりがモデル。
・地域別産物
豊富な農作物，海産物で豊かな地域。
・世界に特化した文様，文字，模様
特になし。

・風景とヒーローのイラスト
暑すぎない暖かいすごしやすい気候。豊富な農作物と海産物。青い海。緑が美しい丘陵。そんな環境ではヒーローだってのびのびと育ちそう。ここはやんちゃで陽気で明るめのヒーローを設定（図1）。

●世界設定（ヒロインの場合）
・時代設定（技術）
ヒーローと同じ。
・時代設定（交通）
険しい山の上に築かれた王国に住む。他国との交流がほとんどない。通信手段もナシ。ほかの国の情報はほとんど入らない。
・国と人種の違い
国内ですべて自給自足でまかなえているうえ、人種が統一されているので、外部の人間に対する拒否反応が強い。
・宗教
神道に近い宗教が存在。信仰心が厚く，宗教の最高権力者は政権すら握る。
・地形と気候の設定
高地特有の岩が多く，乾燥して寒い気候。冬は雪で閉ざされる。
・地域別産物
ヤク，麦など。
・世界に特化した文様，文字，模様
宗教に特化した世界観，文様，文字が存在。

●ヒロイン
閉鎖的な地方にひっそりと生きている，美少女。神秘的な能力がある。見た目はおとなしく口数少ないが，芯が強く，いざってときには大胆な行動ができる（図2）。

●時代設定（技術）
時代設定によって各種の加工技術度が服に反映されてきます。中世ヨーロッパみたいな時代にするのなら，電気動力や蒸気エンジンみたいなのもありませんから，その時代に製作可能なものなどに限界があります。布では現在の化繊みたいな布地は存在しないでしょう。そうなると，ストッキングといったストレッチ素材を使った服装は不可能です。毛にはある程度ストレッチ性がありますが，これも限界があります。体にフィットし伸び縮み自在のボディコンシャスな服装は中世時代に登場させるには無理があります。

さらにファスナーといった服装を固定するアイテムはどうでしょうか。ファスナーがない時代は，ボタンやひも，留め金，バックルなどで固定していました。ゴムなんかはどうでしょうか。ゴムが実用品として普及したのは18世紀後半からです。となると，ばりばりの中世時代にはゴムで固定する服は登場しません。もちろんファンタジーは架空の世界を作り出すわけですから，中世の世界をそっくりそのまま真似する必要はありませんが，剣や弓しか使えない設定なのに，ファスナーやゴムや化繊の服が登場するのはちぐはぐな感じになります。

布を切り縫い合わせる縫製の技術は，日本の着物や古代ギリシャの衣装のように長方形の布を糸で縫い合わせる和裁手法から，体のラインにあわせて布を切った後に縫い合わせる洋裁手法が普及しています。こちらは時代が進むとともに服の形やサイズにバリエーションを出せる洋裁裁断に変化する傾向にあります。しかし着物がいまだに着られていることを考えると，時代で統一するのではなく，地域によって統一するか，あまり厳密に考えなくてもいいかと思います。

近日は縫製技術も布の技術もめちゃくちゃ進歩しております。三宅一生のデザインでよく見られるアコーディオンのような布，糸を使わずに接着剤で縫い合わせた服，まるで本物のような手触りと光沢を放つフェイクファー，蛍光塗料を塗り込み特殊なライトの下で魅惑的に光る服など。近未来のファンタジーを設定するのなら，現代に負けないくらいのアイデアの服で勝負するのもひとつの手でしょう。

作り出した世界の技術力を想定しながら，その世界で作成可能な服装を考えるのはなかなかこだわりがあって楽しいものです。

●ヒーロー
・時代設定（技術）
ここは体にフィットする立体裁断が普及している土地ということで，ジャニーズ系を意識して体のラインが出る感じの服にしちゃおう。おねーちゃん受け対策である。立体的といってもストレッチには限界があるから，ぴったりした系は避ける（図3）。

●ヒロイン
・時代設定（技術）
ちょっと年齢不詳の神秘的な雰囲気がほしいので体のラインは出したくないところ。だから和裁系の服にしてみよう。おにーちゃん対策がちょっと不

図1

図2

図3

図4

備だけど，ま，いいや。ここはゴージャスにシルクでラインにさらっと感を出す（図4）。

・時代設定（通信/交通）

通信や交通手段の普及も服装に反映されます。たとえば，現在のように通信や交通網が発達している世界では，服装の情報も世界レベルで伝達され，流行も世界規模で発生します。したがって，世界のいたるところで同じブランドを入手することも可能ですし，世界の人々が似たような服を着ることもあります。反対に通信交通網が発達していなかった時代となると，一定の服装が流行るエリアは限られてきます。周りの情報が入りにくい場所ではよその地域の服の情報だって入らないのですから当たり前といえます。

情報が入らなければその地域に特化した服装を身にまとうことになるので国や大陸が変わると服装も全然違ったものになる場合があります。もちろん情報そのものが入りにくいのですから，現代のような服装の劇的な変化（流行の変化）も存在しません。同じデザインの服を先祖代々守っていく傾向になります。

交通網はどんなかたちで服装に影響を与えるのでしょうか。シルクロードは，中国産の絹が発達した交通網を通ってヨーロッパまで渡ったてことで名づけられたのは有名ですね。さぞかしヨーロッパでは中国産のシルクを使ったドレスが流行ったことでしょう。でも，反対にヨーロッパの流行なんかもどんどん中国に入ってきたと思われます。新しいものに敏感なデザイナーがいたとしたら，縫製とか，ヨーロッパの文様，レースといったものを取り入れ，オリジナルブランドとして流行らせたことでしょう。

こういったことを念頭に置くと，交易の盛んな地域同士では，布地，縫製方法，柄といったところに結構共通する部分や，似通った点があったりします。反対に閉鎖的な地域ではその地域限定の独自の服装が登場してきて，これまたその地域に特化したデザインの独創性が楽しいものです。実際に私も発展途上国地域へ旅をすると，民族衣装に出あえる機会が多くなり，奥地へ行けば行くほどその閉鎖的環境による衣装の民族化，地域化に遭遇します。そういった変化に出くわすと，いつでも「異国にきたな～」と肌で感じてうれしくなります。

ゲームでも主人公が苦労して行き着いた国の服装がいままでとはまったく別なものになっていたりすると，そのにくい演出に「にやっ」としたこともあります。

●国と人種の違いそして宗教

服装の流行や外来文化の取り入れ度はその国と人種の性格によって差が出てきます。日本みたいに外国の流行を無節操にホイホイ取り込んで，女子高生でもヴィトンのバッグを持っているミーハーな国がある半面，世界各国の情報が入る時代なのに，自分たちの昔ながらの文化や服装を大切に守っている国々もあります。

そのため，交通や情報網が発達していても服装にほかの国の影響が表れない場合も考えられます。自国の文化に誇りを持つがゆえに他国の文化を取り入れるのを拒否する頑固でお堅い国，自国の文化に自信があり他国を認めつつも自国の文化をキープできる余裕のある国，独裁者が治める恐怖国家などが他文化を拒否する傾向にあるでしょうか。あとはその国に住む人種の性格や，過去他国から侵略を受けた等の歴史も影響してくると思います。そういったひと癖ある国や国民性を表現したいのなら，交通や情報伝達手段が整っていても，服装にもある種の法則や規制を施すと現実味を帯びてきます。

また，同じ国の中に住む人種でも住む村によって違う服が登場する地域もあります。これは，服によってその人物がどの村の出身かを明確にしたり，村伝統の服を大切に守ったりしている場合に起こります。国が同じでも村によって服装がガラッと変わったら，習慣とか考え方がぜんぜん違うといった演出も楽しいものです。

宗教も服装に多大な影響を与えるといえるのではないでしょうか。私は過去にイスラム圏とヒンズー教圏に行ったことがあります。服装や生活習

慣に関する掟が非常に厳しいイスラム圏は宗教の定めるところの服装に統一されていて，国全体がその宗教色に染まっていて，国民一同が制服をまとっているという印象がありました。服を見ただけで，宗教が人に与えるもの凄いパワーを感じたわけです。

ヒンズー教圏はバリ島に行ったときのことです。バリヒンズーは一般的なヒンズー教とはちょっと変わっている部分もありますが，ここでは神様の前では肌を露出する服はいけないといった決まりがあります。信仰深い現地の人たちは生活の中にもサルーンといった腰巻きみたいな民族衣装をまとっています。おねーちゃんがサルーンという腰巻きの民族衣装姿でバイクに跨り，頭に神様のお供物を持って走り去る姿には心底驚きました。その服装も決まりの範囲内でおしゃれをする人間がいるので，服の模様や色はかなりバラエティに富んでます。制限の範囲でおしゃれを楽しんでいるのにはちょっと余裕があって色味的におしゃれを楽しんでいる感じでもあります。

特定の宗教を設定したときに，そしてその宗教の決まりや国民性によって，服装にも一種の制限が発生します。神様への信仰心が服装まで影響を与えるなんていまの日本人には考えられない話ですが，現実にあるものなんですね～。

●ヒーロー

・国と人種の違いと宗教

宗教より冒険に憧れる時代。そして一攫千金を狙う国民性。船乗りや商人は信仰心が結構強いが服装に表れるほどではない。それでも，ヒーロー君は冒険時にお守りを持つ程度のことはする（図5）。

●ヒロイン

・時代設定（交通，通信）

ほとんど外界の情報が入らない地で生活しているので，服装は周りの人と同じになる。縫製技術も入ってこないわけだから，昔ながらのものをずーっと引き継ぐ場合が多い。

・国と人種の違いと宗教

過酷な環境ゆえに，環境を神様の力と置き換え，宗教を重んじる。厳密な掟が存在し服装もそれに特化したデザインがある。なので，設定を聖職者にしちゃって周りの人と多少デザインの違った服を身に着けさせることにする（図6）。

●地形と気候

皆さんももちろんご承知のことと思いますが，地形と気候が服装にいちばん影響を与えます。寒い地域だと厚手の服となり，暑い地域だと薄手の服となる。それは当たり前です。日本の気候は暑さ寒さの変動が激しいとはいわれますが，まだまだ世界各国の気候に比べると，あまっちょろいのが現状です。

日本の感覚で世界の厳しい地域の服装を設定すると登場人物たちが死ぬ思いをします。たとえば，寒さ。ヨーロッパ北部やカナダでは冬は激寒状態なので，普通の毛織物のコートでは現地の寒さに対抗できない場合があります。そんなときに重宝するのが皮や毛皮製品だと聞きます。ダウンジャケットや性能のいい化繊が出てきた現代ですと，毛皮ナシでも十分すごせるようになっているのでしょうが，そんな素材がなかった時代を設定するとなると，寒い地域の冬は通常の毛織物のコートでは対抗できません。そのため毛皮は必須アイテムとなるのです。もちろんそういった寒い地域では暖かい皮や毛皮を持つ動物が多くなります。これはきっと，需要と供給のバランスが取れているのでしょう（？）。

反対に暑さはどうでしょう。私がエジプトに行ったときですが，日本の薄着ではエジプトの暑さに対抗できないのがわかりました。砂漠には日陰がなく頭をおおう帽子が必要であり，肌が露出している部分は照らされた日光によって熱を持ったり乾燥してきたりします。激暑の昼間とは反対に，夜や明け方ともなると吐く息が白くなるほど冷え込みます。日本の冬と真夏が１日にして訪れるエジプトでは，日光から日を遮り，明け方の寒さを防ぐ布地の多いズルズル衣装が最適なのです。

そのほかにも高温多湿の地帯では虫を避けるために長袖長ズボンだったり，その場に立ってみて初めて現地の人の服装がいかにその土地にあっているかを認識できます。舞台を設定したときに自分もその場に実際に立っている光景を思い浮かべると，その場にふさわしい服装が出てくるでしょう。

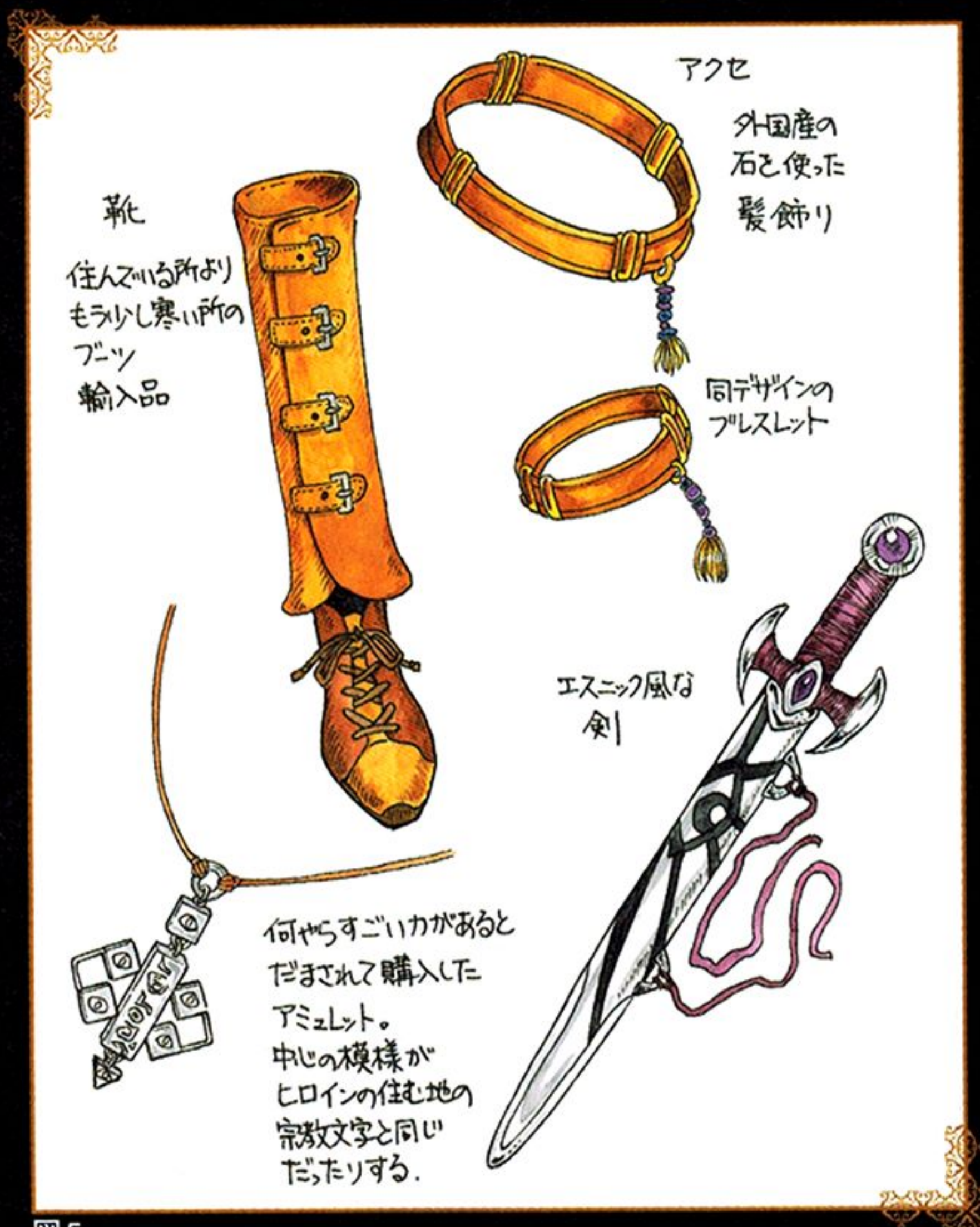



図5

図6

●地域別産物

地域によって入手可能な生地や染色の材料も性質が変わります。日本みたいに四季が豊富で気候がよく農作物に恵まれている土地なら，綿，毛，絹，麻といった繊維が入手できますし，赤，青，黄色といった染料になる植物も豊富です。日本古来の着物はデザインは似通っていますが，色や柄のバリエーションも豊富です。また，木綿のような庶民的な素材から高級シルクを使った高級なものが登場するなど，素材とアレンジのバリエーションは豊富なほうといえるでしょう。

しかし，地域が変わると服装に使える素材に制限がある場合があります。たとえば高地などです。生える植物に制限が出ると服装に植物を回す余裕はなくなります。そのような地域になると，服には動物の毛を使うほうが手っ取り早くなります。たいていの高地では生活のためにヤク，山羊，牛といった動物を飼い，その乳や肉や毛の恩恵を最大限に受けて生活しています。なので服も毛織物が中心となってきます。もちろんそのような地域になると染色に使う植物の入手も困難でしょう（場所によっては鉱物や虫を染料にする地域もあります）。染色の余裕がなければ，色だって取れる動物の毛のみの限定となってきます。

たとえば，白，茶，黒のみといった感じです。これらの色の毛しか入手できない環境では，布のバリエーションがほしい場合には布を織る時点で模様にバリエーションを持たせたり，織る時点での糸の配置で変化をつけます。そのため，織物の模様が豊富になってきます。あとはデザインで勝負でしょうか。服装に使う素材があまり入手できない過酷な環境を設定し，交通や通信の制限をした土地で色彩鮮やかな，どハデな衣装が登場するのはよほど身分がある人か，どこからか輸入した布を使ったという見方が自然です。

図8

図7

鎧なども地域別に変化が出ます。ヨーロッパには全身を鋼鉄で覆う鎧がありますが，日本製の鎧には金属に加え竹を組み合わせたものがあります。これはヨーロッパでは鉱物の産出量が多く，日本ではヨーロッパほど豊富に鉱物が取れなかったためであるといわれます。また，日本の気候上多く取れ，丈夫で軽くてしなやかな竹は鎧としても最適だったのでしょう。さらに高温多湿の日本では金属の錆を防ぐために漆を塗ったりします。これも日本の気候と土地柄の生産物で生まれたものです。その地域で取れるものが限定されてしまう厳しい地域を設定するのなら，服装で厳しさを表現するのも手かもしれません。

●ヒーロー

・地形と気候

暖かい地域だけど日焼けや熱射病の心配はないので，露出度の多い？　服をデザイン（おねーちゃんユーザーの受け狙い）。こういったエリアは半袖でも長袖でもすごしやすい理想的なエリアなので，ほかの人が厚着をしていてもあまり違和感はない。みんな一緒じゃないと心配になる日本ではあまり見かけないが，個人主義が徹底している外国ではよく見かける光景。

・地域別産物

貿易が盛んなことと，気候が温暖なことがあり，豊富な資源でさまざまなバリエーションの布地を使った服が登場する。だから，ここあたりは特に制限を考えず無差別でデザインしてもOK。ミラノコレクションみたいに個性豊かで豪華絢爛，品揃え豊富な世界を意識してもいいかもしれない（図7）。

●ヒロイン

・地形と気候

寒い地域冬の間はまったく身動きが取れない，暖房用の燃料がそんなに確保できないといったことを考慮して，毛皮を一部取り入れる。んでも，これは金を使い放題の聖職者専用の贅沢品のひとつ。

・地域別産物
気候の厳しい高地にあるので，取れる農作物に限定をする。服は毛織物が主。染色もほとんど存在しない。なので，モノトーンの織物の模様で勝負。ここはいっちょうヤクの毛織物をメインに構成するのが自然。でも，色とかをあまり生真面目に限定するのは見た目にもつまらなくなるし，ヒロインが目立たないと演出不足になってしまう。ここは周りが白い世界なので，思い切って赤を取り入れる。鉱物の染料があるんですよーんとか，特殊な虫の体液で染めました〜と逃げるのも手ではある。どんな場合だって逃げはあるので，楽しく設定することを優先（図8）。

●世界に特化した文字，模様，文様
服にはデザインや色，パターン以外に，文字，唐草とか動物をかたどった模様，家柄を表す文様とかが服の模様として登場することがあります。
こういった文様などには，単なるパターンではなく，意味を成している場合があります。たとえば地位や年齢，職業によって使える模様を限定している場合です。単純な例ですと，獅子は王様の服のみ利用が可能，王様以外の王族は鷹，そして貴族はイノシシの文様を身につけなさいと決められている場合とかです。確かにそういった方法ですと，ひと目でその人の地位権力がわかりますし，権力者が服装で自分の力を誇示することもできます。王族以外にも，職業別，住む地域別，出身した家とかで使える文様が変わったりすることもあります。こんなふうに文様は単純な装飾ではなく，意味があって用いられていることがあるのです。
文字はさらに意味を伴って使われるケースが多いみたいです。現実的な例ではジャンバーにメーカー名を入れて，その人物がどの会社に所属しているかをわからせるものから，仏教の梵字みたいに一文字で特定の仏様を表しているのもあります。さらには「寿」「喜」と縁起をかつぐ場合に登場する文字もあります。こういったふうに，オリジナルのファンタジーワールドに特殊文字や宗教があるのなら，そういった模様や文字を服装に登場させるのはどうでしょうか。せっかく作った文字や文様が現実味を帯びてくると思います。

●ヒーロー
・世界に特化した文字，模様，文様
貿易商が多い土地だといろいろな国の言語があふれているのが普通。しかしここでは貿易商の息子の服装とのことで，実家のイニシャルをアレンジした文様をマフラーにひとつ入れることにする。これでぐっとファンタジーっぽくなるというもの。んで，冒険心あふれるワンパク坊主の服装ができあがり（図9）。

●ヒロイン
・世界に特化した文字，模様，文様
聖職者の特定の地位にあるので，宗教に特化した文字と神様の使いとされる聖獣の文様を服装に取り入れてみよう。周りの背景にも同様の模様を用いると統一感があって素敵。これで冒険心あふれるワンパク坊主を引き寄せるための神秘的少女の服装ができあがり（図10）。

服装と世界設定の接点をいろいろと説明しましたが，あまり緊張して時代考証とかにこだわってしまって，自由な発想ができなくなってしまうのも困りものです。ユーザーはあくまでゲームを楽しもうとしているのですから，設定にこだわるあまりにつまらない服装デザインになってもいただけません。最初は楽しく自由に世界設定とか服装デザインをしてから，その次にこだわりを持った設定を服装まで持ち込むと，世界を作り出す自分も楽しいでしょうし，その世界を体験するユーザーも作者のこだわりや細部にわたる気配りを楽しめると思います。

それではゲームクリエータを目指す皆様，こだわりの設定を考えて，素敵なファンタジーゲームを世に放って，私を存分に楽しませてくださいね。

図9

図10

特別企画／*Xbox*と*Indrema* あるいは *Nvidia*の考察

満を侍して発進したはずのPlayStation2がつまずいているあいだに海外勢力が力を伸ばしてきた。
時代はすでにポストPlayStation2に向けて動き出している。
ついに姿を現し，話題を独り占めにしつつあるマイクロソフトの戦略兵器Xbox，
そしてオープンアーキテクチャで伏兵となるか？
Indrema Entertainment System。
最先端のPCコンポーネントを流用しつつ，
さらに高性能なデバイスを目指すこれらのゲームコンソールのポテンシャルはどれほどのものだろうか？
ソニーPlayStation2や任天堂GameCubeとはどの程度違うのか？
そして彼らはなにを目指しているのか？
風雲急を告げる新世紀のゲームデバイス戦争はすでに水面下で次の一手を狙った活動を開始している。
ここでは各種資料から新たなゲームデバイスたちの実像に迫ってみたい。

黒船襲来 Xbox&Indrema

Nakano Shuichi　中野修一

年明けにWinterCESでマイクロソフトのゲームコンソールXboxが発表された。すでに2000春号で概要は少しお伝えしているが，その後(2000年春)発表された内容とはズレも大きく，最近になって明るみに出た情報もあるのでここで一度まとめておきたい。また，Xboxと似たコンセプトのエンタテイメントシステムIndremaについても紹介しておこう。

Xboxと Indremaのスペックは表1のとおり。PlayStation2，GameCube，Dreamcastと PlayStationを併記してみた。

CPU性能ではXboxが頭ひとつ抜け出している。IndremaのCPUがx86系で600MHzということ以外不明なのだが，Xboxの開発にはインテルが絡んでいることなどからAMD製品ではないかという見方もある。逆にAMD製品と決まっていればメーカー名を隠す必要はないという意見もあるのだが……(コスト的にはDuronが本命か？)。いずれにせよ，OSの違いも含めて考えるとIndremaがもっとも高速となる可能性もある。大した差ではないだろうが。

Xboxのアメリカ国内での人気は上々で，ちょっと意外なくらい盛り上がっている。PCでは揺るぎない基盤を持ちつつも，これまでゲーム機ではことごとく日本製品にしてやられていたところへ，PC系の技術をつぎ込んだマシンで一挙に逆転の気配となりビルゲイツは一躍ヒーローとなっているのだろう。

WinterCESの会場では2本のゲームデモ映像が流されたがそれに使用しているのはXboxのパワーのわずか1/5程度にすぎないという(グラフィック部分だけに着目するとおそらくそうなるものと思われる。CPUなどまで考えるとかなり疑問。166MHz程度のマシンで大丈夫かというと無理そうな気がする)。

Xboxの中枢はCPUというよりはXchip，NV2AとMCPXにある。これらはNvidiaが開発しているチップセットで，グラフィック機能統合型ノースブリッジとDSP内蔵のマルチメディアサウスブリッジだ。

そもそもNvidiaにチップセットなど作れるのか？　という疑問を持つ人もいるかもしれない。しかしNvidiaにはSGIでVisualWorkStationを作った連中がごっそり移っているのでさほど心配はない。Cobaltチップですでに実績は残している。VWSはPCではないが，内容としてはPCよりよくできたPCといってもいい代物だ。ちなみに，NvidiaはXchipのグラフィックコアをGeForce2MXにグレードダウンしたような低価格統合型PC用チップセットも計画している(コードネームCrash)。また，サウスブリッジもインテル用

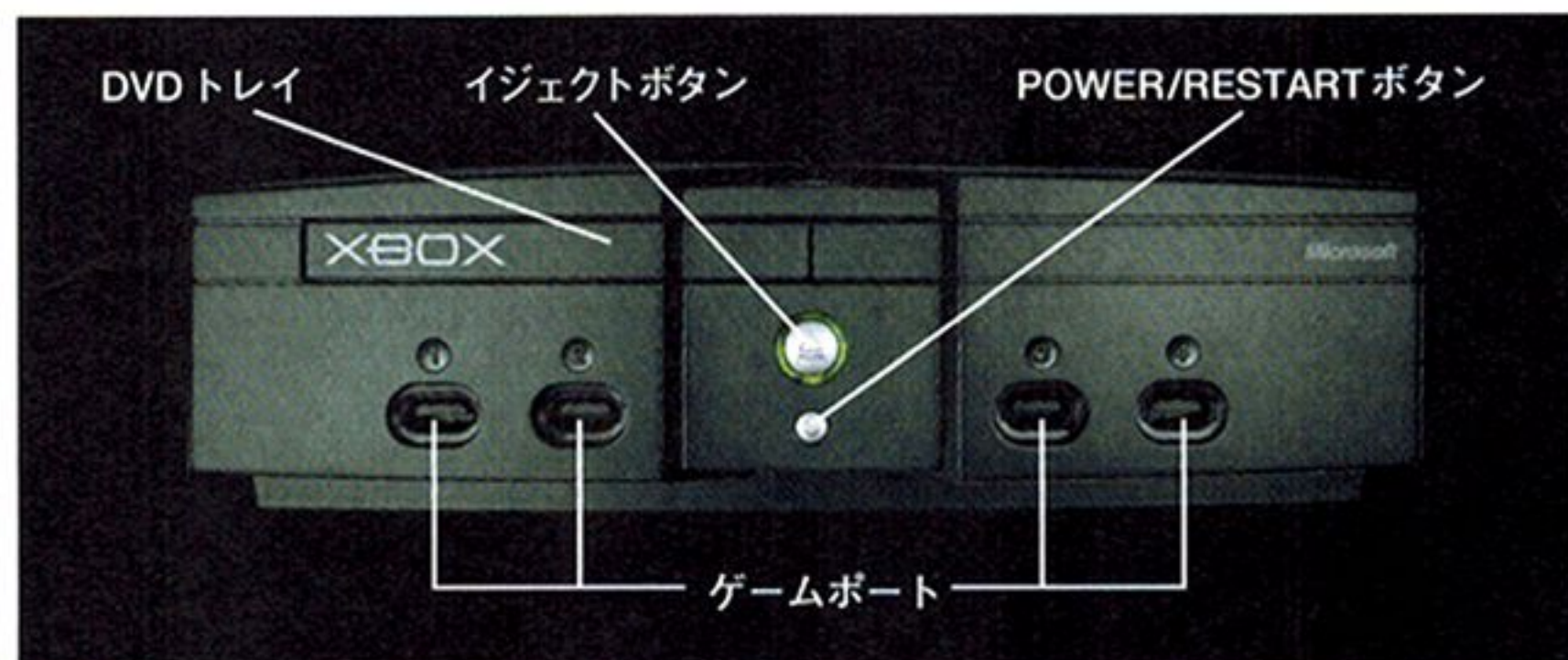



図1 **Xbox正面図**
コントローラはUSBをベースとしたインタフェイスで接続される。データ転送速度は12Mbpsだ。POWERボタンは軽く押すとRESET動作，押し続けると電源OFFとなる仕様だ。イジェクトボタンは外周部が黄色と青色に光る仕様となっている。DVDビデオの再生には別売のリモコンが必要だ。

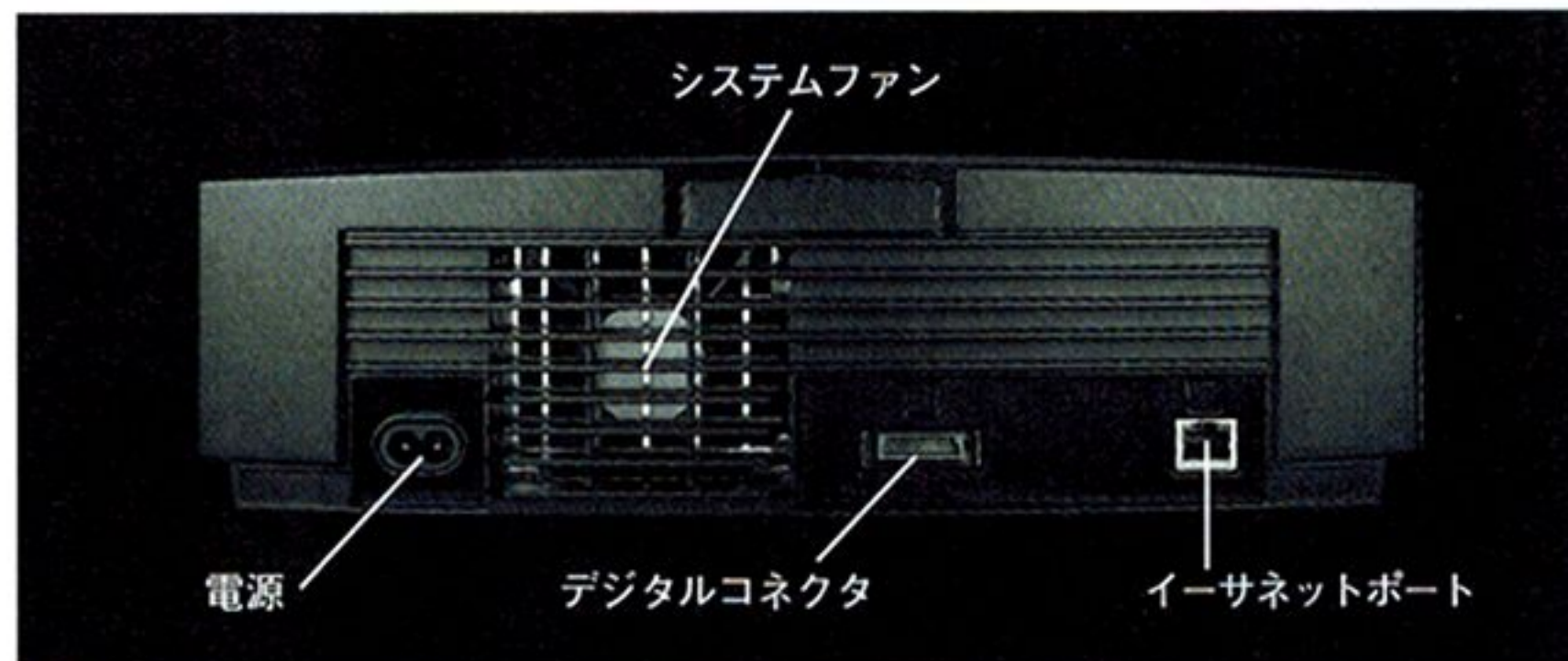



図2 **XBox背面図**
イーサネットポートは100Base-TX仕様。デジタルコネクタからはAV関係の信号がまとめて出力されており，ケーブルで分岐することになる。対応したケーブルを使うことでD端子やDTS出力なども可能となる。

［表1］各種ゲームコンソールのスペック

	PlayStation	Dreamcast	PlayStation2	GameCube	Xbox	Indrema
CPU	**R3000カスタム**	**SH4**	**EmotionEngine**	**Gekko**	**PentiumIIIカスタム**	**不明**
種類	32ビットMIPS	32ビットSH	128ビットMIPS	32ビットPowerPC	32ビットインテル	32ビットx86
動作クロック	33MHz	200MHz	300MHz	400MHz	733MHz	600MHz
メモリ	**EDO DRAM**	**SDRAM**	**DRDRAM**	**1TSRAM+SDRAM**	**DDR SDRAM**	**DDR SDRAM？**
容量	2MB	16MB	32MB	40MB	64MB	64MB
システムメモリ帯域	0.13GB/s	0.8GB/s	3.2GB/s	1.6GB/s	6.4GB/s	5～10GB/s
ビデオ	**GPU**	**PowerVR2DC**	**GraphicsSynthesizer**	**Flipper**	**NV2A**	**NV20**
メモリ容量	1MB	8MB	4MB	3MB	統合	32MB
動作クロック	33MHz	100MHz?	150MHz	200MHz	250MHz	300MHz?
ビデオメモリ帯域	0.13GB/s	1.6GB/s	48GB/s	12.8GB/s	6.4GB/s	19.2GB/s？
公称ポリゴン性能	0.36Mpoly/s	3Mpoly/s	66Mpoly/s	6～12Mpoly/s	125Mpoly/s	120～180Mpoly/s
サウンド	**SPU**	**AICA**	**SPU2**	**Flipper**	**MCPX**	**不明**
チャンネル数	24チャンネル	64チャンネル	48チャンネル	64チャンネル	256チャンネル	不明
サウンドメモリ	0.5MB	2MB	2MB	8KB	統合	不明
OS	**なし**	**WindowsCEカスタム**	**なし**	**不明**	**カスタムWindows**	**DV Linux**
記憶装置	**CD-ROM**	**GD-ROM**	**DVD-ROM**	**SingleDVD?**	**DVD-ROM**	**DVD-ROM**
HDD	なし	なし	なし	なし	8GB HDD	10GB HDD
メモリカード	128KBメモリカード	128/512KBメモリカード	8MBメモリカード	512KBメモリカード	8MBメモリカード	なし

左トリガー
右トリガー
X，Yボタン
START，BACKボタン
左アナログ
スティック
A，Bボタン
右アナログ
スティック
デジタル
スティック
STARTボタン
BACKボタン

図3 **Xbox用コントロールパット**

NVIDIA XGPU and MCPX
Powering Microsoft's X-Box
64MB DDR Memory
733 MHz CPU
NVIDIA XGPU
TV
• Breath-taking 3D Graphics
• HDTV Output
Game Pads and Expansion Slots
NVIDIA MCPX
Audio
Communications
Storage and Media
• Mind Blowing 3D, Dolby Digital™ Audio
• Broadband, 10/100 Mb Ethernet, 56K Modem
• 5x DVD and 8GB HDD

図4 Xboxのブロック図

MCP1とAMD用MCP2のリリースが予定されている。今後はチップセットメーカーとしても押さえておかなければならない存在だ。

CPU，ポリゴン性能，音楽機能など，Xboxはあらゆる面でPlayStation2の3倍の性能を目指して開発されたハードウェアだけに，下手に使ってもPlayStation2くらいの絵は出てくるというのは心強いだろう。

なお，ほぼ同等の構成のハードにも関わらずIndremaのほうは知名度自体がほとんどないに等しい。しかしどちらもNvidia製品をキーデバイスとして使っているというのは重要だ。3dfxを吸収し，Nvidiaはグラフィック界の巨人として君臨しつつある。

Xboxではグラフィック的な表現力も申し分ないといっていいだろう。PlayStation2では理論上可能（大半はソフトウェア処理だろうが）なフィーチャが発表時のスペックとしてたくさん掲げられていたのだが，それらの大半はまだ実用段階とは思えない（実際ゲーム画面でお目にかかれないし）。それらはXboxではすぐにでも実現可能なものであり，さらに遙かに進んだ機能も提供される。

むしろ，ハードウェア性能自体は単なる目安にすぎない。真に恐るべきは，そこに宿るDirect X8というシロモノである。XboxはDirectXを動かす箱（ただし理想的な）にすぎないのだ。

DirectXという剣

最初のうちはDirect Xなど，もの笑いの種だった。だが，統一的なインタフェイスをみんなで作っていこう，という方向は思想的に正しい。Windows95で成功したマイクロソフトには必要な求心力があった。当時，グラフィックでは最先端を進んでいたSGIなどはDirect3Dを詳細に調べあげ，「まだまだだね」としつつも，その後は急速にマイクロソフトに接近していっている。現在では共同開発者の位置を確保してしまった。DirectXはバージョンを更新するごとに確実に洗練された内容となってきており，いまではあらゆる分野の最先端を切り開く側の存在になろうとしている。

さて，「画質」の違いとはなんだろうか。

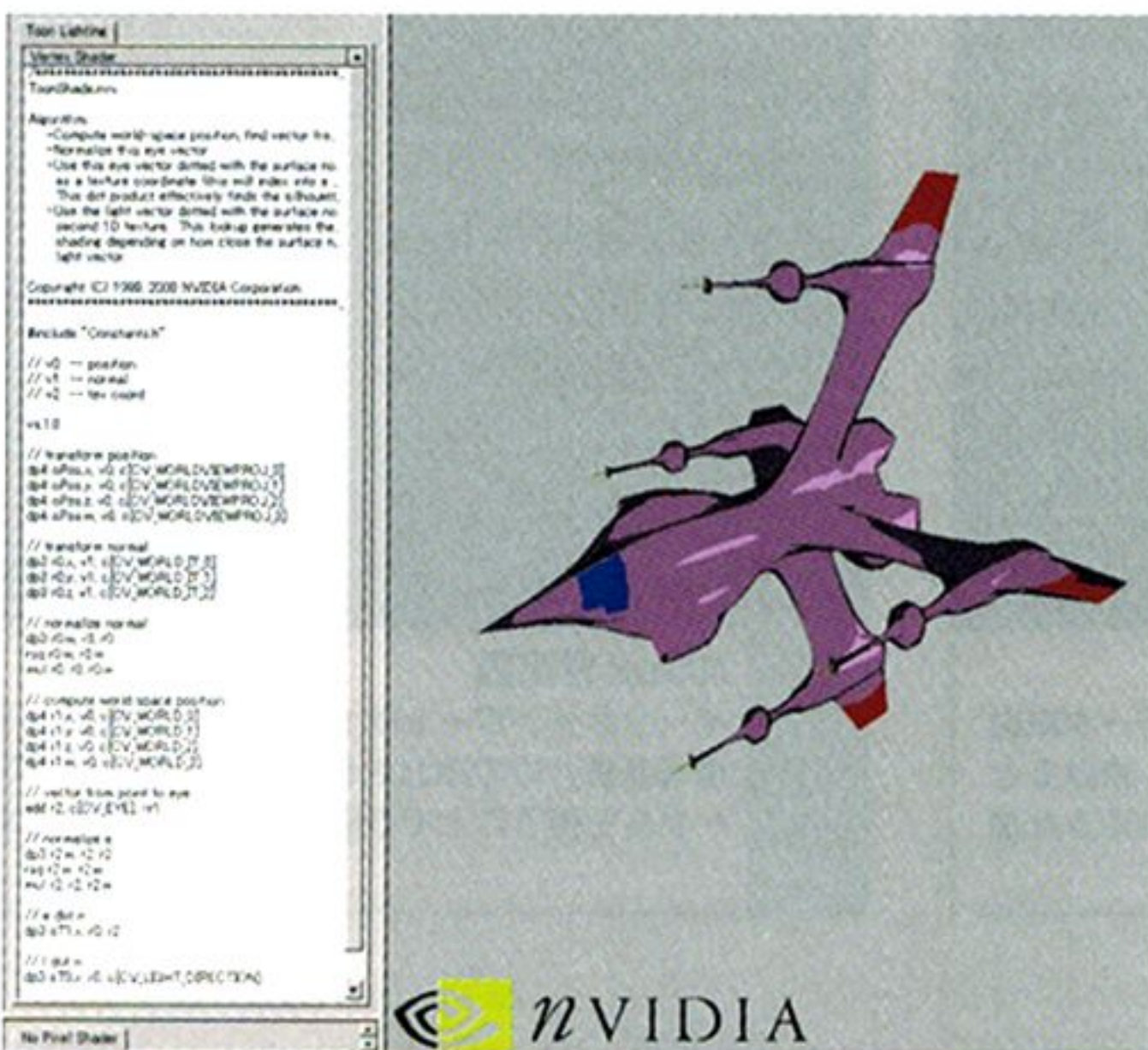



図5 トゥーンシェイダーとそのコード

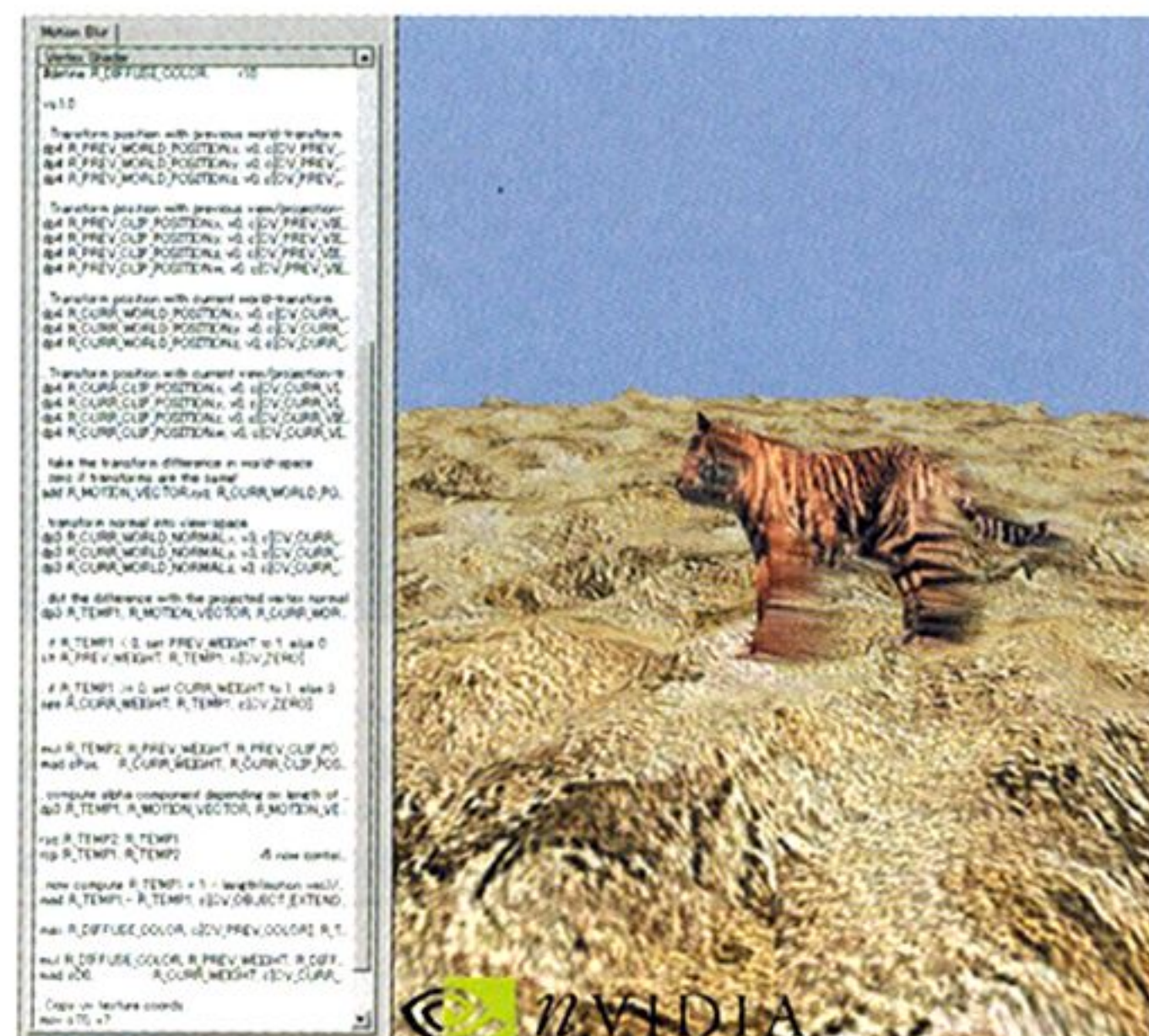



図6 VertexShadingによるモーションブラー

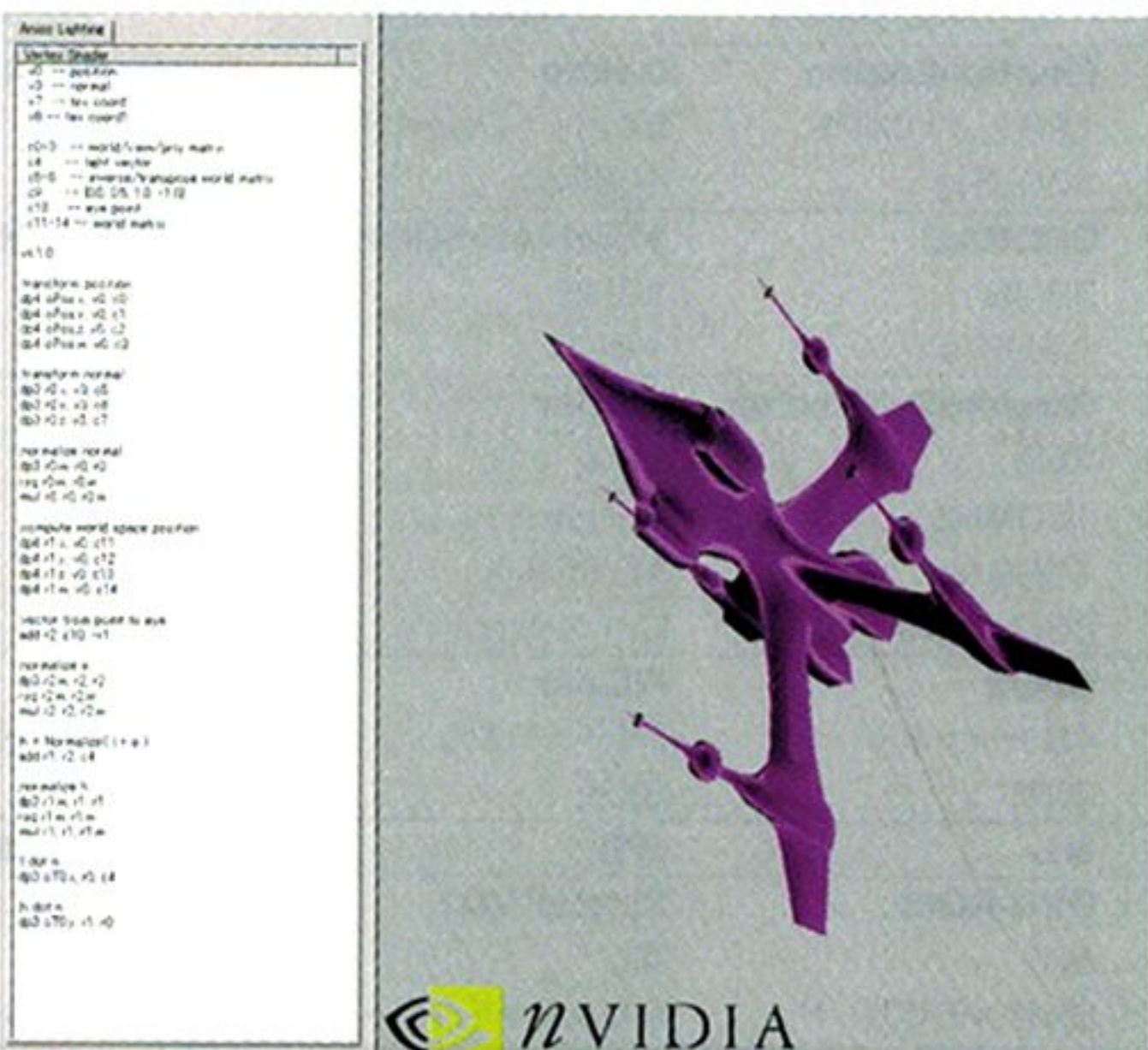



図7 異方性反射によるベルベットのような輝き

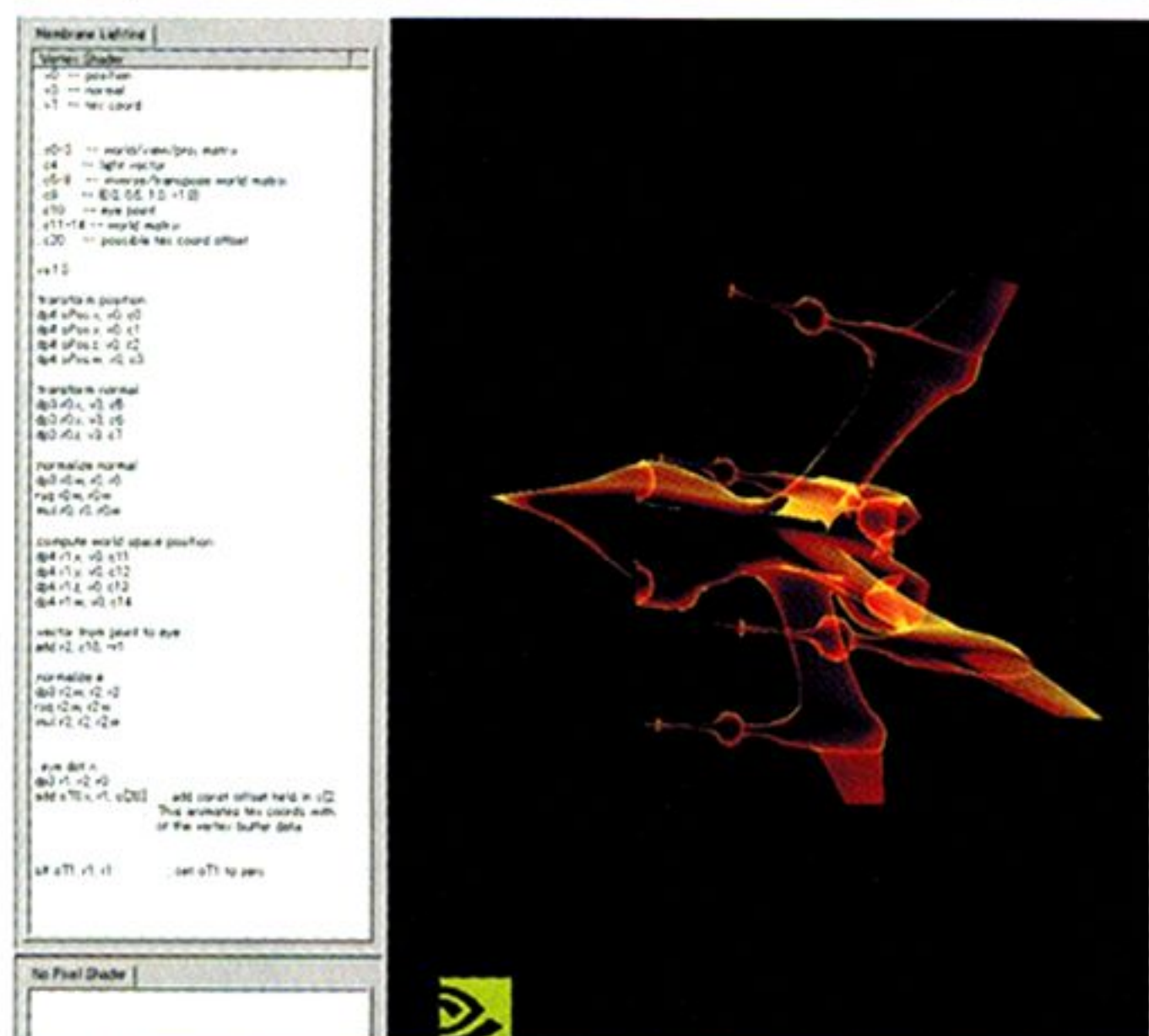



図8 メンブレンシェイダー

いろいろな要素があるのだが，安定感。自然さといった指標にまとめられるのではないかと思う。

安定感を出すのがアンチエイリアスであり，バイリニアフィルタ，トライリニアフィルタといった不足する解像度や離散的な情報量を補間しようというアプローチだ。これで，画質は変わるか？というと，圧倒的に変わるといって差し支えないだろう。特にテレビ画面のような低解像度時には威力を発揮する。

自然さは，ひと言ではいいづらいが，たとえば，PlayStation世代のマシンでは，人物などでふっくら丸まるとしたグローシェーディングの質感があった。腕など断面が丸いものはまだしも，手の甲なども丸まるとして見えたゲームは数知れないだろう。これって変だ。

それはポリゴン数を増やすことで解決できる。またはシェーディングモデルを変更することでも解決できる。逆にいえば単純なシェーディングモデルの違いはポリゴン数で近似的に代替できる。だが，単純でないものについてはそうはいかない。これは少し説明が必要だろう。

たとえばPlayStation2では基本的にグローシェーディングでしかない(フラットができないという意味ではない)。

じゃあXboxはフォンシェーディングなのかというと，すでにそういう単純な話ではない。

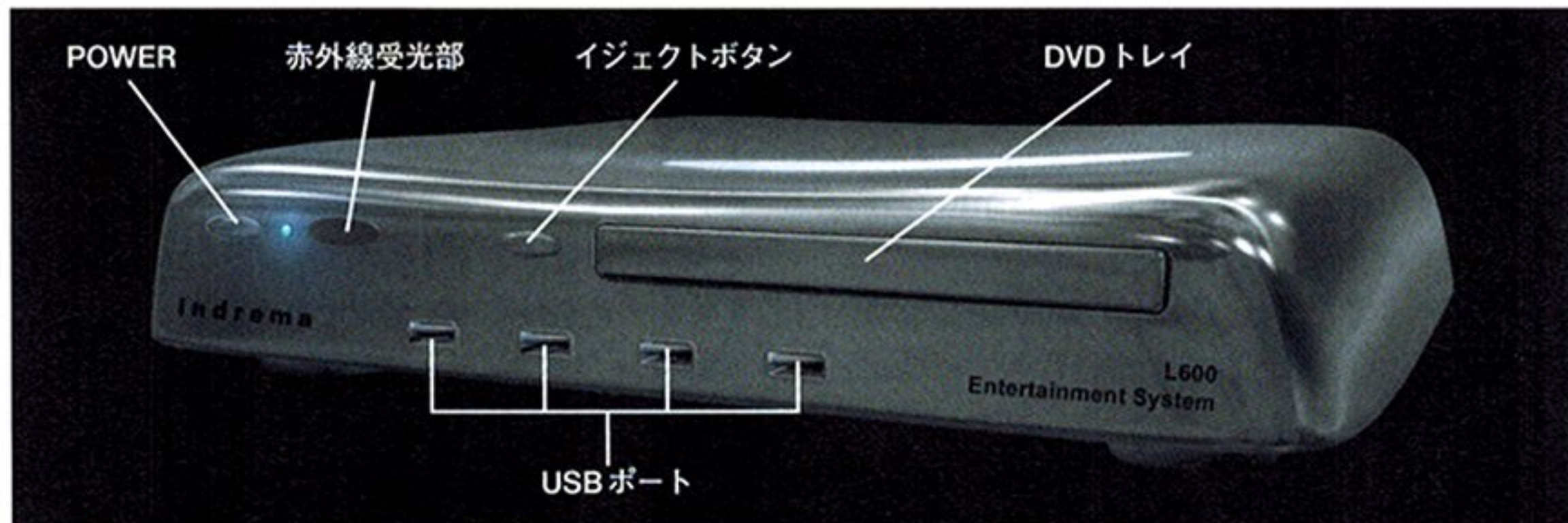



図11 **Indrema Entertainment System L600**
Indremaの最初の製品がこれ。標準タイプのUSBが採用されているのでPC用のデバイスなどもそのまま使用できそうだ。標準マウス/キーボード（オプション）はワイヤレス仕様だ。

Phongの式を使ったものがあるかどうかはわからないので正確にフォンシェーディングとはいえないが，少なくとも法線は補間可能だろうし，もっと複雑な演算をドット単位に行えるようになってしまった。NvidiaのデモではBllinシェーディングされたベジエ曲面の鏡面反射体(バンプつき)というのが上がっている。残念ながら現状のチップではハードウェア実行はできないのだが(画面はリファレンスラスタライザによる出力)，次世代のDirectX8対応のチップではこういうことが平気でできるようになってしまうのだ。

これらがいわゆるVertexShaderやPerPixel Shadingという奴なのだが，ピクセルレベルの演算を行うPerPixelShadingに対して，VertexShadingというのはわかりづらい。これは頂点の座標情報や法線，色など，頂点に関連したさまざまな情報を操作できるものだ。頂点座標をいじればポリゴンの変形や運動を表現でき，モーフィングのようなこともできる。かと思えば，法線情報を変えて異方性反射を行ったり，ポリゴンの色をさまざまに変えてしまうこともできる。こういうのをシェーディングと呼んでいいのだろうか？

これらは描画時にT&Lとは別の演算ユニットで処理され，頂点演算を行ったり，まさにピクセルを書き込むそのときに一定量の演算処理を施すことができるというものだ。プログラマブルとはいえ，ループや分岐ができない一直線の処理だけだが，あっちのデータを取ってきてこっちのデータと演算してそっちに書き込んで……といった処理が最初からレンダリングパイプラインに組み入れられているというのは強力だ(負荷はそれなりにかかるが)。ハードウェアサポートされていない場合は悲惨なことになる。Bllinシェーディングのサンプルはリファレンスラスタライザだと Athlon/1GHzのマシンで5秒で1コマがやっとだ(1280×1024で最大化時)。

参考までにNvidiaのデモの例を挙げておくが，Direct X8に準拠という

図12 Indrema開発用のWebページ

図9 PixelShaderによる鮮鋭化

図10 Bllinシェーディング。Cubeマップ部はたった3行

ことはこういうのがグリングリン動くと考えても間違いではない。Xboxの表現力は非常に高度なものとなる。プログラマブルな表面素材は，ゲーム機の「グローシェーディングでディフューズばりばりっス」といった感じの画一的な質感とは次元が異なる効果をもたらす。

これらはPC用ビデオカードでもサポートされるようになるが，最初から搭載されていることを前提にソフトを作ることができるとなると，出てくるソフトの画質レベルもかなり違ってくるだろう。

DirectX自体は今後も進化を続けていく。ほかの記事にもあるとおり，1年で互換性が吹き飛んでしまうくらい変わっていく。そうなるとXboxの仕様はどの程度Fixされ続けるのかという点が問題になってくる。Xbox2のような規格は出てくるのか？　ソフトウェアだけバージョンアップするようなこともちろん可能だ。どのような対応が取られるのか注目したいところだ(もの凄く気が早い話だが)。

Indrema

Indremaは正確にはIndrema Entertainment System(IES)というx86ベースのゲームコンソールでXboxの発表と前後して発表されていた。ハードウェア仕様的にもかなり近い感じだ。おそらく日本に紹介したのはNetXがいちばん早かったのではないかと思う。記念すべき紹介文を全文引用しよう。

「なんじゃこりゃ？」

以上だ。

DirectXがXboxの強みだと書いたが，DirectX用に加速されたハードウェアを流用しておいしい思いをすることだってできる。それがIndrema流だ。確かにDirectXだと楽な部分というのも大きいが，そうでなければ扱えないわけではない。

Xboxがその名のとおりDirectXベースのコンソールであるのに対し，IndremaはOpenGLベースである。OpenGLという描画APIだけでは間にあわない部分ではOpenStreamとかOpenALといったオープンアーキテクチャが採用されている。中心となるのがDV Linuxである。XboxがDirectXの箱なら，Indremaはオープンアーキテクチャの箱なのだ。

なんとなくかつてのDirect3D対OpenGl論争を思い出すが，DirectXはDirectDrawやDirect3Dだけじゃなくてたくさんの機能を備えている。そこでOpenGLだけではなくてほかのオープンアーキテクチャで補われたシステム，それがIndremaのバックボーンである。OSのカーネルからドライバレベルに至るまでほとんどがオープンソース。WebブラウザはもちろんGeckoベースだ。ゲームコンソールでこれだけ開かれた環境というのは空前絶後かもしれない。

製品の発売はまだだが，すでにIndrema開発環境は配布されている。開発に必要な環境は，

- **CPU**　x86プロセッサ(600MHz以上)
- **HDD**　10GB以上の空き
- **ビデオカード**　Nvidia製(GeForce2GTS以上)
- **OS Linux**　カーネル2.4(2.2以上)

となっている。ビデオカードあたりがちょっと痛いところだが，PC用上で比較的簡単に組めるシステムでそのまま開発が行える。開発環境でさえカーネル2.4以上が前提となっており，普通のLinuxシステムとはちょっと様子が違うことがわかる。

SDKは，

http://idn.indrema.com/iesdk/

からダウンロード可能だ。

もっとも注目すべき点は，Indremaではフリーソフトウェアの開発，配布が可能だということだろう。デベロッパー登録や審査は必要になるが，これまでのゲーム機ではなかなか難しかったものが実現されている。

Direct3D対OpenGL

3DAPIとしてはOpenGL(MESA)が使用されることはすでに述べた。OpelGLのAPIをオープンソースで実装したMESAが採用されている。Direct3Dと比較してどうなのだろうか。

OpenGLベースのゲームというとQuakeやUnrealといったFirstPersonタイプしか思い浮かばないので偏った印象を受けがちなのだが，もちろんそういったものしか作れないといった制約はない。なお，3dfxのGlide APIもMESAを基本にしたものらしい。

UnrealやQuakeIII ArenaなどのシステムはPCゲーム上でも最高の質感を持ったものとしても知られているように，OpenGLベースだからといってスペック的にDirect3Dに遅れを取ることはない(少なくとも現状では)。むしろいまだにQuakeなどで十分研究してからDirect3Dに導入してるような節もある(Direct3Dでのポリゴンをベジエ補間するデモなどで，なぜかQuakeのキャラクターが使われていたりする)。

NvidiaのテクニカルデモはOpenGLで発表されていた。最近はNVeffectBrowserが開発されたためか，Direct3Dベースのものが多くなってきているとはいえ，チップとOpenGLとの親和性は決して低くはない。

MESAはオープンソースのOpenGL API実装であり，きわめて現実的な仕様といえる。MESAでゲームを作成しておけばほかのシステムへのポーティングも難しくはないだろう。

ほかのオープンソース環境についてもまとめておこう。

●DV Linux

OSのDV Linuxは民生用機器に最適化されたLinuxのインプリメンテーションでマルチメディア機能などを強化されている。IndremaがRed hatと協同で開発しているものだ。ただ，夜中に勝手にバージョンアップするぞ(カーネルの再コンパイルとか勝手にやるわけだろうか？)というシステムもちょっとどうかなという気もしないではない。

●OpenAL(http://www.openal.org/)

オーディオ関係のオープンAPIだ。3DサウンドなどをR実現するためのAPIでクロスプラットフォームで同じ仕様のプログラムが使えるようになる。現在はストリームバッファの実装作業中だそうだ。

なおIndremaのサウンド関係は明らかにされていないが一応それなりのものは載ってきてそれなりにスゲーと書いてはある。OpenALはCreativeLabsが中心に進めているので，EMU10Kあたりが載ってくると仮定すると，64ボイスでEAX搭載，まあこんなところかなというスペックにはなる。

●OpenStream

ストリーミングを扱う。ビデオやマルチメディアデータをLinuxで扱えるようにするための拡張で，Direct Rendering Interface(DRI)を使って高速な処理を実現している。

●Xtrema

Xfree86 4.0.1に最適化されたGUI環境で，Degital Rights Managementシステムをを備え，著作権管理なども取り扱う。

その他のハード仕様で映像出力だけでなく入力端子も各種備えていたりHDTVデコーダ，MPEG2関連もデコーダだけでなくハードウェアエンコーダが搭載されているというのが凄い。それにしてはHDD容量が貧弱だが一応拡張可能となってはいる(方法は不明)。

テレビチューナが入っているかどうかは不明だが(CATVアダプタやHDTVデコーダは入っている)，そのままでデジタルビデオレコーディング機能を備えている。ビデオを2ストリームまでサポートしているというのはテレビ録画しながらDVDが見れるという感じでなかなか応用範囲が広い。

基本的にIndremaは単なるゲーム機というのではなくて，「FutureTV」という単語が示すように，従来のテレビを置き換えるものというコンセプトで売り出している。セットトップボックスのような印象も強い。

Indremaではもうひとつ革新的な機構が採用される。それはGPUスライドベイで，グラフィックチップをアップグレードできるシステムだ。Nvidiaのビデオチップは半年ごとに3割以上の性能向上ペースを保っているので，いくら優れたチップだからといって，ほっておけば陳腐化するのは避けられない。NV20クラスのチップであれば基本バス性能には当分問題がない(256ビットDDR)と思われるので，チップ部分のみ交換可能にするというのもある程度現実的だろう。なお，交換用カードは複数のベンダーから50～100ドル程度で発売されることになるだろうとされている。

その他細かく書いてないが，ネットワーク関係をはじめいろんなものを搭載している。ただ，この仕様のまま，299ドルでちゃんと出てくるのかというのが疑問ではある。最新のFAQでは2001年春発売となっているのだが，ちょっと怪しい気はしている。

世の中の動向

DreamcastはSH4，PowerVR2DC，その他周辺が1チップ化されてPC用の周辺機器として100ドル程度で2001年夏に発売されるとの見通しも出ている。秋にはノートブック用コンポーネントも出ると噂されているが……。さて？

考えてみれば，Xbox，Inderma，そしてPC用ビデオカード。それらがともにNvidiaチップをキーとしている。NvidiaはSGIから大量の人材を得て急成長してきた。さらに3dfx(というかその一部門のGigapixel)を吸収してさらに力をつけている。GigapixelというのはSGIでReality Engineなどを作ってた人たちが超低価格で同等のものを作ろうとしてた会社だと思っておけば間違いない(Xboxであわや採用されるかというところまではNvidiaと競りあったといわれている)。

任天堂のGameCubeはArtX(現在はATIの傘下)の技術によって成り立っているが，ArtXもSGIからのスピンアウト組である。

時代はSGIが動かしているといっていいのだろうか？

なお，マイクロソフトは来年にはMoBoxを発表するともいわれている。これはGameBoyとWindowsCE機の中間的な存在で，Xboxソフトが動作するモバイル端末であるという。確かに，Nvidiaは今後のメイン市場をモバイルに向けて動き出しているというが……。

Step to the Black Arts

PCのプロセッサはGHzのクロック単位になる時代がやってきた。
1GHzのAthlonマシンを組み上げ，EX68を久々にアップデートした。基本環境を作ってZ-MUSIC ver.3.0をセットアップ。mpcm.xを組み込み，サンプル曲のQuackmarchを流す。たちまちあたりはノイズに包まれる。再生周波数を変えたり，いろいろ試すとノイジーながらもようやく曲が聞こえてきた。こういうのもなんとか動くようにもなってきてるのかと少し感動。さらにいろいろやっても改善がないので，EX68の設定を確認すると，おっと，クロック制限をつけたままだった！　10MHz制限から一気に無制限に設定。おお，まぎれもなくあの曲だ。今度は曇りもなく聞こえてくる。
かつてOh!X編集部に，この調子っぱずれな曲が初めて鳴り響いたときのことを思い出す。AD PCM変調というX68000でも最高難度の技を駆使し，悲願であったPCM音源制御を実現した。このポルタメントはまさに勝利の響きだ（一晩中鳴らしていたらさすがに苦情がきたけど）。
EX68もよく動いている。
PentiumⅢ/666MHzでは少しノイズが乗ることがあるがAthlon/1GHzではほぼ大丈夫。ついにこういうのまで動くようになってしまったというのはうれしくもあり，一抹の寂しさも覚える。
ハードウェアはどんどん高速に多機能になっていく。ハード側のパワーでソフト側の問題を覆い隠せるというのもある意味事実だ。しかし，だからソフトウェアがタコでもよいということにはならないし，よくできたソフトウェアは人を感動させうるものだ。最近はそんな体験もすっかりなくなったところに，意外なところからハードの力とソフトの力のコンビネーションを感じた。これはやはり勝利のメロディなのだろうか。

Macintoshでゲームを作ろう！

ねおだ 如　Neoda Nyo

ホビープログラミング用の資料に乏しいMacintosh。ここではMacintoshの代表的な開発環境であるCodeWarriorを使って，ゲームの制作を進めていきます。Macゲームでのもっとも基本になる処理を順に解説していきますので参考にしてください。

ホビープログラマがMacintoshでゲームのプログラムを作ってみようと考えたとき，まず問題となるのは参考にできる資料と助言してくれる人の少なさでしょう。

なにか作りたいと思って手をつけようとしても参考になるものは，ほとんどなにもありません。

とかくMacintosh用のC言語のための資料は量が少なく，「ちょっとなにか作ってみよう」などと考えても，必要な資料が載っている本は探すのが大変です。パソコンの量販店やら専門店やらを渡り歩いて，ようやく見つけた参考書籍はレベルが高すぎて結局役に立たなかったり，「これは！」と思って手に取ったら，Windows用の書籍だったり……なによりつらいのは，教科書的な良書はあるにも関わらず，「ゲームを作ってみよう」とか「○○してみよう」というプログラムの実践的な本になると，ほぼ皆無だということです。

さらに，掲示板やメーリングリスト，果てはオフ会などに参加してMacプログラミングのエキスパートと目されている方たちに自分の疑問，質問をぶつけてみても，「基本的なことですから，Inside Macintoshを読んでください」「公開されている情報ですから，英文資料を読んでください」「仕事じゃないし……」とか答えられたりして……。

そんな壁に阻まれてそれ以上前に進むことができなくなってしまった経験をお持ちの方，少なくないんじゃないでしょうか？

かくいう筆者もまた，そんななかのひとりだったりします。

そこで今回は，Mac開発環境のデファクトスタンダードともいえるMetrowerks社のCodeWarriorを使って，ごく簡単なゲームを作成し，それを通してC言語を用いてMacintosh上で開発を行うためのヒントや，いままでに得ることのできた経験則（ノウハウといえない……）をなんとか公開していこうと思っています。

作ったゲームの速度の問題や説明を簡単にするために，対応機種はPowerPC搭載機種とし，またC言語のだいたいのところは理解しているものとして進めていくことにします。この点はどうかご了承ください。

最近はFutureBasicやREAL Basicなど，学びやすく扱いやすい開発言語も登場し，Carbon APIなどを介したOS Xへの移行もあって，大きな変革を遂げようとしているMac開発環境ですが，いままでの経験が一夜にしてまったく無駄になってしまうことは考えにくく，むしろこれからなにかを作ろうとするときによい指針になってくれるはずです。

CodeWarriorについて

CodeWarriorはMetrowerks社が開発，販売する統合開発環境です。Macintosh，Windowsをはじめ，多くのOSに対応していますが，最近は特にPalmなどの各種携帯端末やPlayStation2などの家庭用ゲーム機のクロス開発[※1]に力を入れているようです。

CodeWarriorには，ソフトを作ることはできても基本的にそれらを公開することが許されないDiscoverProgramingと，開発・公開ともに問題のないProfessionalの2種類があります[※2]。Professionalパッケージは実売7万円近くで，REAL BasicやFuture Basicと較べてもダントツに高価で気軽に買える値段ではありませんが，できることもいちばん多く，できればなんとか手に入れたいところです。

もちろん，本音としては，Professional相当の開発環境と公開可能なライセンスをもっともっと安価に（できれば，いまのDiscoverと同程度の価格で）入手できるのが理想なのですが……なんとかなりませんかね？

ともかく，最近のApple社のサンプルコード[※3]も，その多くがCodeWarriorのプロジェクトファイルか，あるいはCodeWarriorで読める形式として提供されており，これからMacintoshのプログラムを始めるのであれば選択肢のひとつとしてよいと思います。

このほか，C/C＋＋が使える開発環境としては，Apple社の開発ツールMacintosh Programmers Workshop（MPW）が有名です。これは現在フリーで公開されており，ダウンロードして使用することができます[※4]。Apple社自身の開発ツールということもあり，使いこなすことさえできればこれがいちばんいい選択肢かもしれませんが，MPWは英語版のみで関連文書もすべて英文。お世辞にも使うための環境が整っているとはいえません。

これからMacintoshプログラムの勉強を始めようと考えているのであれば，資料を探す手間と時間などを考えて，まずは日本語の資料が揃っているほかの開発ツールを選んだほうが無難でしょう。

今回のプログラムでは，このCodeWarriorのバージョン3を使用しています。現在，CodeWarriorはバージョン6が発売されていますが，そのバージョンで新規プロジェクトを作ってCD-ROMに掲載されているプロジェクトのソースを張り付ければ問題なく動くでしょう。

※1　クロス開発とは，実際に動かしたいマシン（ターゲット）とは別のマシンでソフトの開発を行うこと。たとえば，Macintosh上で，WindowsMe用のアプリケーションを開発することで，携帯端末やゲーム機用のソフトを作るときにこの方法が使われたりします。
※2　Macintosh用のC言語入門書などに添付されているLite版は，新規プロジェクトおよび新規ファイルの作成を行うことができません。また，680x0コードのみを生成することが可能です。したがって，残念ながら今回の記事の内容をそのまま使用することはできません。
※3　アドレスは，http://developer.apple.com/samplecode/です。
※4　アドレスは，http://developer.apple.com/tools/mpw-tools/です。CodeWarriorにも付属しています。

やるなら気楽にシンプルに

今回のプログラムは，ごくごく簡単なシューティングゲームです。名づけて，SimpleTextならぬSimpleShot。

出てくるのは自機と敵機の2つ，左右に移動して撃ちあうだけ，弾は敵味方それぞれ1発ずつ，効果音はなしという，単純この上ないシューティングゲームです。さらに話を簡単にするためにアップルイベントやメニューなどには一切対応せず，起動後，マウスクリックで終了させることにします。

かなり乱暴な方法ですが，「メニューやイベント，ウィンドウの操作やマルチタスクなどに対応しなければならない」というMacOS上でプログラムするうえでのルールはひとまず棚上げにして，ともかく「とりあえずでも動くものを仕上げる」ことを目標にしてみましょう。

さて，これで処理しなければならないことをかなり減らすことができましたが，それでもMacOS上でシューティングゲームを作る以上，避けて通れない点もいくつかあります。それらを以下に挙げてみます。

1)　ウィンドウを表示する
2)　表示したウィンドウにキャラクターを表示する
3)　プレイヤーからのキー入力を受け付ける
4)　マウスがクリックされたら終了する

このほかにも，キャラを動かしたり，当たり判定を確かめたりする必要がありますが，上の4つのポイントはMacOSプログラミングの要となるToolBox[※5]を扱う必要があるものばかりで，きちんと理解しておく必要があります。逆に，これらは今回作ろうとするSimpleShot以外にもいろいろ応用することのできる，もっとも重要な部分のひとつです。ここを押さえておけば，MacOS上でCプログラミングの，特に描画に関する部分が少しずつ見えてくるはずです。覚えなければならない点も少なくありませんが，マイペースでやっていきましょう。

※5　ToolBoxとは，MacOS上でプログラムを作る際に必要になってくる関数群。その関数の使い方をまとめたのがInsideMacintoshという本です。このInsideMacintoshの日本語訳なんかが素早く出てくれれば，もう少し楽ができるのになぁ……。このあたり開発関係の書籍が充実しているWindowsコミュニティが羨ましいところです。

えらく小さな第一歩

まず最初にsmallest.cを見てください。

ただ単にシステムビープ音を鳴らし，マウスクリックを待つだけのものです。えらく小さなプログラムですが，これでも立派なMacintoshアプリケーションになります。本当にこんな小さなプログラムコードでアプリケーションが作成できるのか，CodeWarrior上でのプログラムの手順を追いながら確かめていくことにしましょう。

1）プロジェクトの作成

最初にCodeWarrior IDEを起動し，プロジェクトを作成することにしましょう。

ファイルメニューから「新規プロジェクト」を選択するか，Shift＋Command＋Nを押すと，プロジェクトステーショナリ選択ダイアログが表示されます。ここでは図1のように，MacOS＞C_C＋＋＞MacOS ToolBox＞MacOS ToolBox PPCを選んでください。「フォルダを作成する」のチェックを入れておくのを忘れずに。プロジェクト名はなんでもかまいませんが，とりあえずSmallest.prjとでもしておきます[※6]。これで図2のようなプロジェクトウィンドウが開いたはずです。

2）プログラムの入力

次に実際のプログラムの入力です。

今度は新規のファイルを作成します。ファイルメニューから「新規」を選択するか，Command＋Nで「名称未設定」という名前のウィンドウが開きます。ここにsmallest.cを入力していくわけです。全部入力したら，まず保存です。ファイルメニューから「保存」か，Command＋Sです。ファイル名はSmallest.cに（もちろん，別の名前でもかまいません）。

このとき，プロジェクト作成時に作られたフォルダ（プロジェクト名をSmallest.prjとしたときはフォルダ名はSmallestになる）に保存するようにしましょう。これが，今回のアプリケーションのためのソースリストとなります。保存したら，もう一度リストを確認してみましょう。打ち間違いがあったら直しておきます。もちろん，手直しはあとでもできます。

3）プロジェクトへのファイルの追加と削除

せっかく入力したソースリストも，そのままではコンパイルすることができません。そこで，プロジェクトメニューから「ファイルの追加」を選び，2）で保存したソースをプロジェクトに追加します。図3のようなダイアログが出てきますので，Smallest.cまたは自分でつけたファイル名を選び，「追加」をクリック，ダイアログの下の部分にいま選んだファイルが移動していることを確認して「終了」をクリックします。

すると図4のようなダイアログが表示されます。両方にチェックが入っていることを確かめて「OK」をクリックしてください。これでプロジェクトへ無事ファイルを追加することができたはずです。

追加したファイルはプロジェクトの選択されていた部分に挿入されますが，これはどこにでも移動させることができます。移動の方法は図2のプロジェクトウィンドウ上にあるファイル名の部分をドラッグするだけです。

次は元からあるファイルの削除です。

これは，簡単で，削りたいファイルをプロジェクトウィンドウ上からクリックして選択し，プロジェクトメニューから「選択項目を削除」を選ぶか，Command＋Deleteとするだけです。

ただ，削除するとはいっても，プロジェクトから外されるだけで，ファイルそのものはちゃんと残っています。間違えて削除してしまったら，もう一度追加してやれば大丈夫。ただ，ライブラリを削除してしまうと厄介なので，間違えて削除しないように注意しましょう。

このプロジェクトからは，SillyBalls.cを削除します。

4）コンパイルとアプリケーションの作成

いよいよコンパイルです。ツールバーかプロジェクトウィンドウにある図5aのボタン（メイクボタン）をクリックすれば，コンパイルとメイクを自動的に行ってアプリケーションを作成してくれます。隣にある図5bの矢印形のボタン（実行/デバッグボタン）は，さらに実行まで行ってくれますが，今回はアプリケーションがちゃんと作成されたかどうか，まず確かめてみることにしましょう。

エラーメッセージも表示されず，コンパイルとメイクが無事終了したら，Finderに戻ってSmallestフォルダを覗いてみます。図6のように「MacOS

リスト1　Smallest.c

```
/* 関数プロトタイプ */
void    main(void);
void    initToolBox(void);

/* メイン関数 */
void main(void)
{
    initToolBox();/*ToolBox初期化*/

    SysBeep(0);/*システムbeepを鳴らす*/

    while ( !Button() ){};   /*マウスボタンが押されるまで待つ*/
return;/*終わり*/
}
/* 初期化 */
void initToolBox()
{
    /*必要な各種マネージャーの初期化。*/
    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(nil);
    InitCursor();
}
```

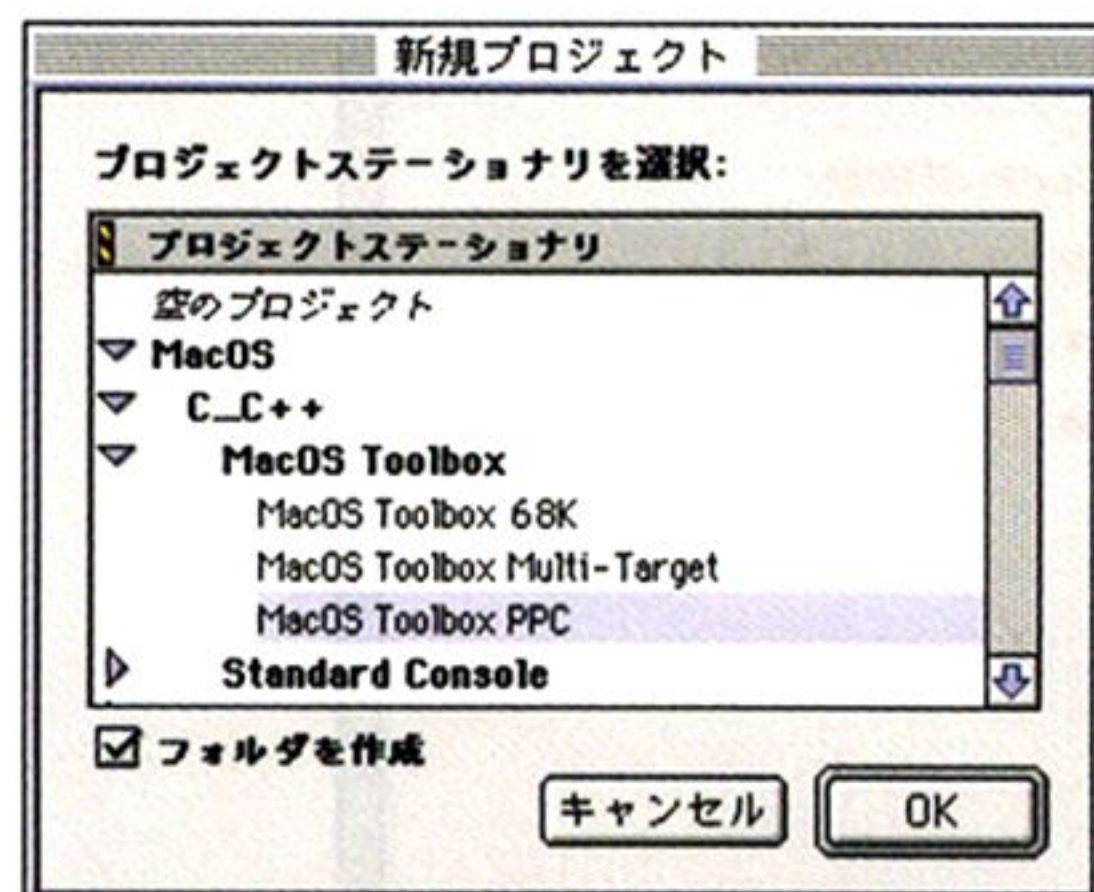



図1

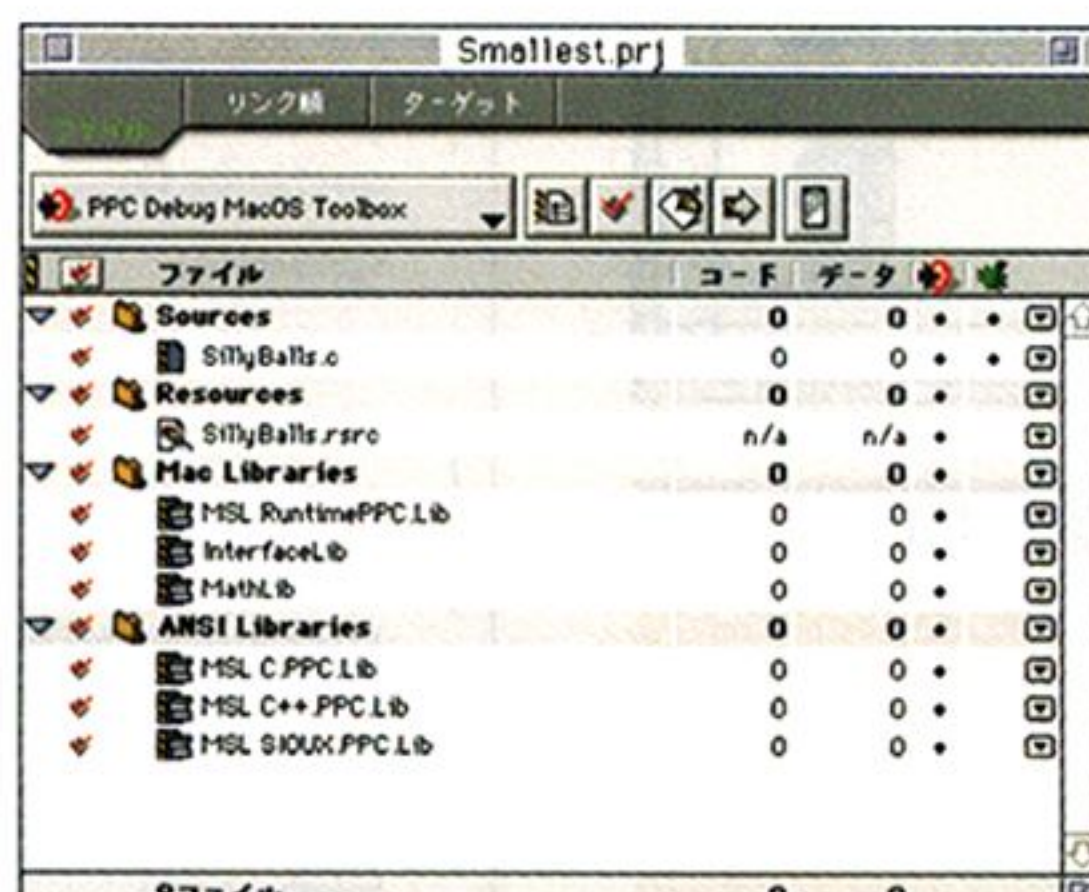



図2

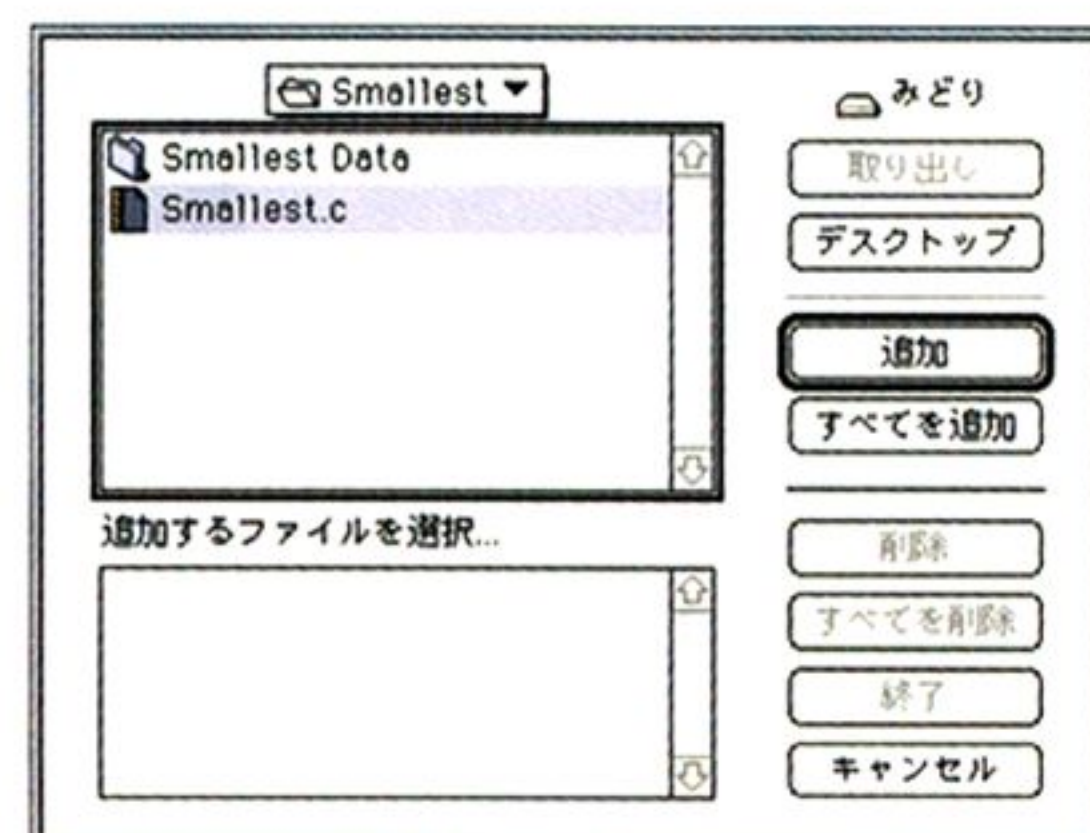



図3

Toolbox DEBUG PPC」という名前のアプリケーションが作成されていたら成功です。

もしエラーが発生した場合，図7のようなエラーウィンドウが開いてどの位置にエラーがあるかを教えてくれます。掲載されているソースリストと自分で打ち込んだリストをよく見比べて確かめてみましょう。図7の場合では，main関数の最後のreturnの後ろに，;（セミコロン）を打ち忘れていますね[※7]。

また3）で，SillyBalls.cをプロジェクトから外すのを忘れてもエラーが発生してしまいますので注意してください。

5）アプリケーションの実行

さっそく完成したアプリケーションを実行してみましょう。

Finder上から「MacOS Toolbox DEBUG PPC」をダブルクリックです。……どうですか？　なにもないメニューバーが出てきて，聞き慣れた警告音（または自分で設定した警告音）が鳴ったはずです。ここでマウスをクリックすれば見事アプリケーションは終了します。

もし，白いメニューバーが表示されたままになって終了しなかったら……Command＋Option＋Escキーでアプリケーションを強制的に終了させ[※8]，もう一度ソースリストを見直してみましょう。きっとタイプミスかなにかがあるはずです。間違いを修正したらメイクボタンをクリックして，再度コンパイルとメイクを行うのも忘れずに。

6）Smallest.cの中身を見る

それでは，Smallest.cのなかでいったいなにが行われているのか順を追って見ていきましょう。

まず最初に，initToolBox ()関数が呼び出されます。これはアプリケーションで使用するToolBoxやインタフェイスを初期化するものです。この関数はほとんど定型句みたいなもので，ほかのプログラムに持っていったとしても書き換えたりする必要はありません。また，プログラムの最初に呼び出しておけば，どのToolBox関数を呼び出すにも安心です。

ただしinitToolBox ()関数内での，各ToolBox関数呼び出しの順番は変更しないようにしてください。順番を変えると，動かなかったり，おもむろに暴走したりすることがあります。

簡単に説明していくと，InitGraf ()関数はQuickDraw，InitFonts ()はフォントマネージャ，InitWindow ()はウィンドウマネージャ，InitMenu ()はメニューマネージャ，TEInit ()はテキストエディットマネージャ，InitDialogs ()はダイアログマネージャをそれぞれ初期化します。また，InitCursor ()はカーソルを初期化します（矢印にして見えるようにします）。

次はSysBeep ()関数です。これはシステムに設定された警告音を鳴らすもので，関数表にあるとおり引数に意味はありませんが，なにか値を入れてやる必要があります。

警告音を鳴らしたあとでマウスクリックを待つのが次のwhile文です。

Botton ()関数は，呼び出された時点でマウスボタンが押されていればtrueを，いなければfalseを返しますので，この場合はマウスボタンを押さない限り条件文はtrueとなり，ループし続けることになります。マウスボタンが押されれば，条件はfalseとなりますので，ループを脱出してアプリケーションが終了します。

こんな短いプログラムではありますが，ちゃんとMacintoshのアプリケーションとして起動し，終了させることができました。これで当初の目標のうち，「マウスがクリックされたら終了する」という関門を突破することができました。

どんなもんでしょう？　ちょっと拍子抜けしてしまった方が多いかと思います。が，一見難しいといわれているMacintoshのプログラムも，こんないい加減なプログラムで結構なんとかなってしまうことだけは理解していただけたのではないでしょうか。

※6　本来，MacOS上では拡張子モドキはつけなくても構わないのですが，検索などを行う際に便利なので筆者はつけるように心がけています。ただし，勝手につけた拡張子モドキが原因でWindowsとのファイルのやり取りを行ったときに変な問題が起きる可能性もあります。

※7　このようにC言語では，たった1カ所のタイプミスのために，数カ所から数十カ所にエラーが発生してしまうこともあります。たくさんのエラーメッセージが出ても驚かずに，先頭から1カ所ずつ確かめていきましょう。

※8　本当なら，このあとすぐに再起動すべきです。この記事ではそういう手間を惜しんで書いていますが……こういうことして暴走してしまったら，自分の責任ですので笑って諦めましょう（笑）。

図4

図5a

図5b

図6

表1　えらく小さな第一歩

```
/*システムビープ音を鳴らす*/
void    SysBeep( short theNumber );
引数　：theNumber　ダミー値。なにを渡しても動作は変わらない
返り値：なし

/*マウスボタンの状態を得る*/
Boolean Button(void);
引数：なし
返り値：関数が呼び出された時点でマウスボタンが押されていればture，押されていなかったらfalse
```

ResEditに手を染めて

CodeWarriorでアプリケーションを作成し，起動と終了がきちんと（？）

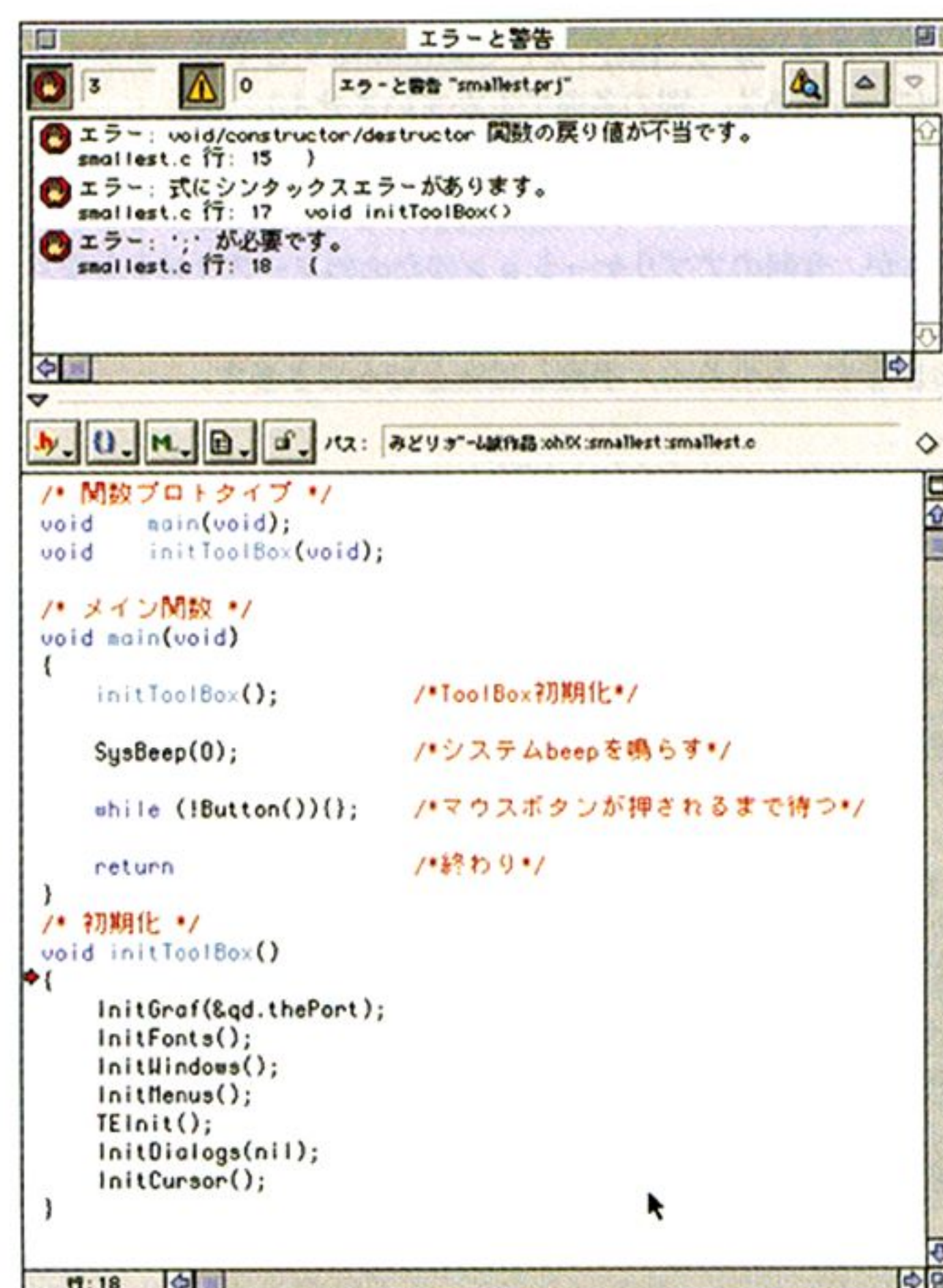



図7

できるようになったわけですが，今度は「ウィンドウを表示する」という関門にチャレンジしてみましょう。リソースエディタであるResEditも登場です。

リストは，window.cです。

なんかsmallest.cに毛が生えた程度ですが，このソースでちゃんとウィンドウを表示させることができます。では，順を追って見ていきましょう。

1）プロジェクトの作成とファイルの追加

さっそく，プロジェクトを作成してみましょう。

プロジェクト名はwindow.prjとしておきます。作成の手順は，前節の1〜3と同じです。ただし，リストはwindow.cとなります。

ここでコンパイルして実行しても，一応アプリケーションが作成されます。起動することもできますが，一瞬だけ白いメニューバーが描画されて，すぐに戻ってきてしまうことになります。これは必要なリソースがないために，GetNewCWindow ()関数がNILを返すためです。NILが戻り値となれば，ExitToShell ()関数が呼び出されてFinderに戻ってしまいます。

ここはちょっと我慢して，次に進みましょう。

2）新規リソースファイルの作成

ResEditを使って，必要なリソースの作成を行います。

もしResEditをインストールしていない方は，この機会に入れてしまいましょう。Macintoshプログラムを作成するにあたっては，ResEditなどのリソースエディタは必要不可欠のものです[※9]。ResEditはCodeWarrior CDに収録されていますので，ファイル検索などで探して，インストールしてください。ドラッグ＆ドロップするだけでOKです。

ResEditを起動するとタイトル画面が出てきます。ここでマウスをクリックすると，図8のようなダイアログが表示されます[※10]。ここで先ほどプロジェクトを作成したときにできたwindowフォルダに移動し，「new」ボタンをクリックして新しいリソースファイルを作成します。ファイル名はwindow.rsrcとしておきます。

3）WINDリソースの追加と変更

ここがwindow.prjのキモの部分です。が，別にたいしたことはしていませんので，ぼちぼち行きましょう。

作成された直後，window.rsrcは空っぽで，いま開いたウィンドウ，空のリソースリストにもなにも表示されていません。そこでResourceメニューからCreate New Resourceを選ぶか，Command＋Kで新しいリソースを作成します。

ここで，図9のようなダイアログが表示され，リソースタイプを選択するように出てきます。今回はウィンドウを画面に表示したいのですから，ウィンドウのためのリソースを追加しなければなりません。そこで，左のスクロールリストの中から「WIND」を選んで「OK」をクリックしましょう。右側のフィールドの中にWINDと入力してもかまいません。

すると，WINDリソースが作成され，すぐに図10aと図10bにあるような2つのウィンドウが表示されます。

図10aのウィンドウはwindow.rsrcのなかにあるすべてのWINDリソースのリストを表示するもので，WINDを追加するたびリストに加えられていきます。また，図10bのウィンドウはWINDエディタで，新しくWINDリソースを作ったり，図10aのウィンドウに表示されているWINDリソースのリストをダブルクリックしたりすると表示されます。

WINDエディタを見てみましょう。左下の部分に，Top，Height，Left，Widthの4つのフィールドがあるはずです。もしHeightとWidthのフィールド名がBottomとRightとなっていたら，WINDメニューから「Show Height & Width」を選択してください。表示が変更されます。

この4つのフィールドのうち，Heightフィールドに384，Widthフィールドに512を入力します。また「Initially visible」のチェックが入っていることを確かめましょう。図10bと同じ状態になっていれば問題はありません。確認したらWINDエディタを閉じます。

この状態でWINDリソースリストは，図10aのように表示されているはずです。ここで，いま作成したWINDリソースを選択して，Resourceメニューから Get Resource Infoを選ぶか，Command＋Iとすると図11のようなリソース情報ウィンドウが開きます。ここでミソとなるのがIDフィールドです。

window.cを見直してみてください。kWindow_IDは，#define文によって200に定義されています。GetNewCWindow ()関数に渡すリソースIDと，ResEditで定義するWINDリソースのIDは同じにする必要があります。そこで，リソース情報ウィンドウのIDフィールドの値を200にします。

IDフィールドを変更してリソース情報ウィンドウを閉じたら，WINDリソースリストを見てみましょう。IDの値が変わっているのが確認できると思います。

IDフィールドの下にあるnameフィールドは，リソースに名前をつけるためのものです。プログラムには関係ない部分ですが，アプリケーションが同じ種類のリソースを複数持つようになると番号だけではどこで使われるリソースなのか見分けにくくなってきます。そんなときのためにリソースに名前がつけられるようになっているのです。名前をつけると，リソースリスト上でも，名前が表示されるようになります。

デスクトップ ▼
あか
みどり
みどり
Eject
Desktop
Cancel
New
開く
Use Alias instead of original

図8

リスト2　window.c

```
#define kWindow_ID 200
#define NIL         0L

/* 関数プロトタイプ */
void main(void);
void initToolBox(void);

/* メイン関数 */
void main(void)
{
    WindowPtr mainWindowPtr;/*Window構造体を保存するためのポインタ*/

    initToolBox();/*ToolBox初期化*/

    /*リソースの情報をもとにウィンドウを作る*/
    mainWindowPtr = GetNewCWindow( kWindow_ID, NIL , (WindowPtr)-1 );
    if( mainWindowPtr == NIL )
    {
        /*失敗したら、なにもせずにFinderへ戻る*/
        ExitToShell();
    }

    SysBeep(50);
    while (!Button()){};/*マウスボタンが押されるまで待つ*/

    return;
    /*終わり*/
}
/* 初期化 */
void initToolBox()
{
/*必要な各種マネージャの初期化。*/
    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(nil);
    InitCursor();
}
```

表2　ResEditに手を染めて

```
/*リソース情報をもとに，新しいウィンドウを生成する*/
WindowPtr GetNewCWindow( short windowID , void * wStrage , WindowPtr behind );
引数：windowID
リソース番号：wStorage window構造体をスタックに置く場合，CWindowRecord型の変数を作って，そのポインタを渡す。普段は0を渡してヒープ内に構造体を作る
：behind ウィンドウの重なりの順番を示す。−1を渡すといちばん手前に，0を渡すといちばん後ろに，いまあるウィンドウへのポインタをWindowPtr型で渡すと，その後ろにウィンドウを作成する
返り値：生成できた場合はウィンドウ構造体へのポインタ，生成できなかった場合はNILが返る

/*プログラムを終了させ，Finderに戻る*/
void ExitToShell(void);
引数：なし
返り値：なし（返らない）
```

これでリソースの作成がひとまず終わりました。

すべてのウィンドウを閉じ，window.rsrcを保存してResEditを終了させます。

4）リソースのプロジェクトへの追加

CodeWarrior IDEに戻って，リソースをwindow.prjへ追加します。

追加の仕方はソースリストのときとまったく同じです。追加したら，元からあるリソースファイルを削除します。

5）コンパイルとアプリケーションの作成

これでアプリケーション作成のための準備はすべて整いました。

前節のSmallest.prjと同じように，ツールバーかプロジェクトウィンドウにあるメイクボタンをクリックすれば勝手にコンパイルとメイクを行ってくれます。エラーが出る場合はソースリストにタイプミスがあるので，確認して修正します。

6）アプリケーションの実行

いよいよ実行です。

windowフォルダ内に完成した「MacOS Toolbox DEBUG PPC」アプリケーションをダブルクリックすれば，警告音が鳴って画面の左上のほうにNew Windowという名前のウィンドウが表示されたはずです。ちゃんと表示されていますか？　表示されていたら，マウスクリックで終了させましょう。

もし一瞬だけ白いメニューバーが表示され，ウィンドウが出てくる前にアプリケーションが終了してしまったら，ResEditで設定したWINDリソースのID番号とwindow.c中の#define文で定義した#kWINDOW_IDの値が両方とも200になっているか確かめてみてください。

それ以外の場合は，ほぼ間違いなくソースリストのタイプミスです。じっくりと確かめていきましょう。

7）window.cの中身を見る

では，window.cの中身を追ってみましょう。

main()関数の最初に宣言されているのはWindowPtr型の変数mainWindowPtrです。WindowPtrというのは，ToolBoxがウィンドウを扱うときに使う構造体へのポインタです。難しいことは置いておいて，とりあえずMacOS上でウィンドウを扱うプログラムを組む場合にいちばんよく使う変数型だと覚えておいてください。

initToolBox()関数は前節で解説したとおりです。変更されたところはありません。

次はwindow.prjのポイントとなる，GetNewCWindow()関数の登場です。これは，WINDリソースの情報をもとにして新しいウィンドウを作る関数で，３つの引数を持っています。

最初の引数は参照するWINDリソースのID番号です。引数として渡したIDと実際に参照したいWINDリソースのIDが一致していないとアプリケーションの動作がおかしくなるので気をつけてください。

２番目の引数は，確保した構造体をどこに置くかを決めるのですが，0を渡しておけば問題ありません。window.cでは先頭で#define文でlong型の0として定義したNILを引数として渡してあります。

３つ目の引数は画面上でのウィンドウの重なり位置を指示します。複数のウィンドウを扱うアプリケーションを作成するのは，ずいぶん先のことになりそうですので，－1を渡します。なにも考えないで，そのまんま－1を渡すとコンパイルした際に「'int'を'struct GrafPort ＊'へ変換できません」と文句をいわれるのでWindowPtr型にキャストしておきましょう。

さて，これで第２の関門，「Windowを表示する」を突破することができました。

見た目は大したプログラムではありませんが，リソースと連携することで，ちゃんとMacOSらしいウィンドウを表示させることができました。

これで，ゲームのための舞台装置はだいたい完成したといってもよいでしょう。残っているのはゲームのための楽屋，キャラの表示とキー入力です。いずれも，いままでのようにはいかない難関です。

ですが，そこに手をつける前に，いままでの復習ついでにwindow.prjをいろいろいじって遊んでみることにしましょう。

※9　最近は，Resourcerというさらに多機能で使いやすいリソースエディタも登場していますが，いかんせん高い（実売２万7000円前後）ので，筆者はResEditを使用してます。
※10　このダイアログが表示されない場合，Fileメニューで「new」を選ぶか，command＋Nで新規リソースファイルを作成してください。

遥かなり，リソースの座

前節で作ったwindow.prjでウィンドウを表示することはできるようになりました。しかし，表示されている位置は中途半端，ウィンドウのタイトルはNew Windowのまま。なんとも具合がよくありません。

そこで，もう少しResEditでリソースをいじってみて，window.prjで作られるウィンドウを見栄えよくしていってみましょう。リソースを操作すれば，そのままアプリケーションに反映されていきます。

実際の動作でそれを確かめつつ，遊んでみることにしましょう。

1）window.rsrcを開く

前節で作ったwindow.rsrcをResEditで開きます。Finder上でダブルクリックするほか，CodeWarrior上のプロジェクトウィンドウのファイル名をダブルクリックしてもResEditを立ち上げて読み込ませることができます。

window.rsrcのリソース一覧のなかには，先ほど作ったWINDリソースだけが入っています。ここから図10aのWINDリソースリストと図10bのWINDエディタを開きましょう。前節で新しくWINDリソースを作成したときのように，自動的に開いてはくれませんので，自分でダブルクリックし

Info for WIND 128 from window.rsrc
Type: WIND　Size: 29
ID: 128
Name:
Owner type: DRVR / WDEF / MDEF
Owner ID:
Sub ID:
Attributes: System Heap / Locked / Preload / Purgeable / Protected / Compressed

図11

Select New Type
TEXT / TMPL / TOOL / vers / wctb / WIND / wstr
WIND
OK / Cancel

図9

WINDs from window.rsrc
ID	Size	Name
128	29	

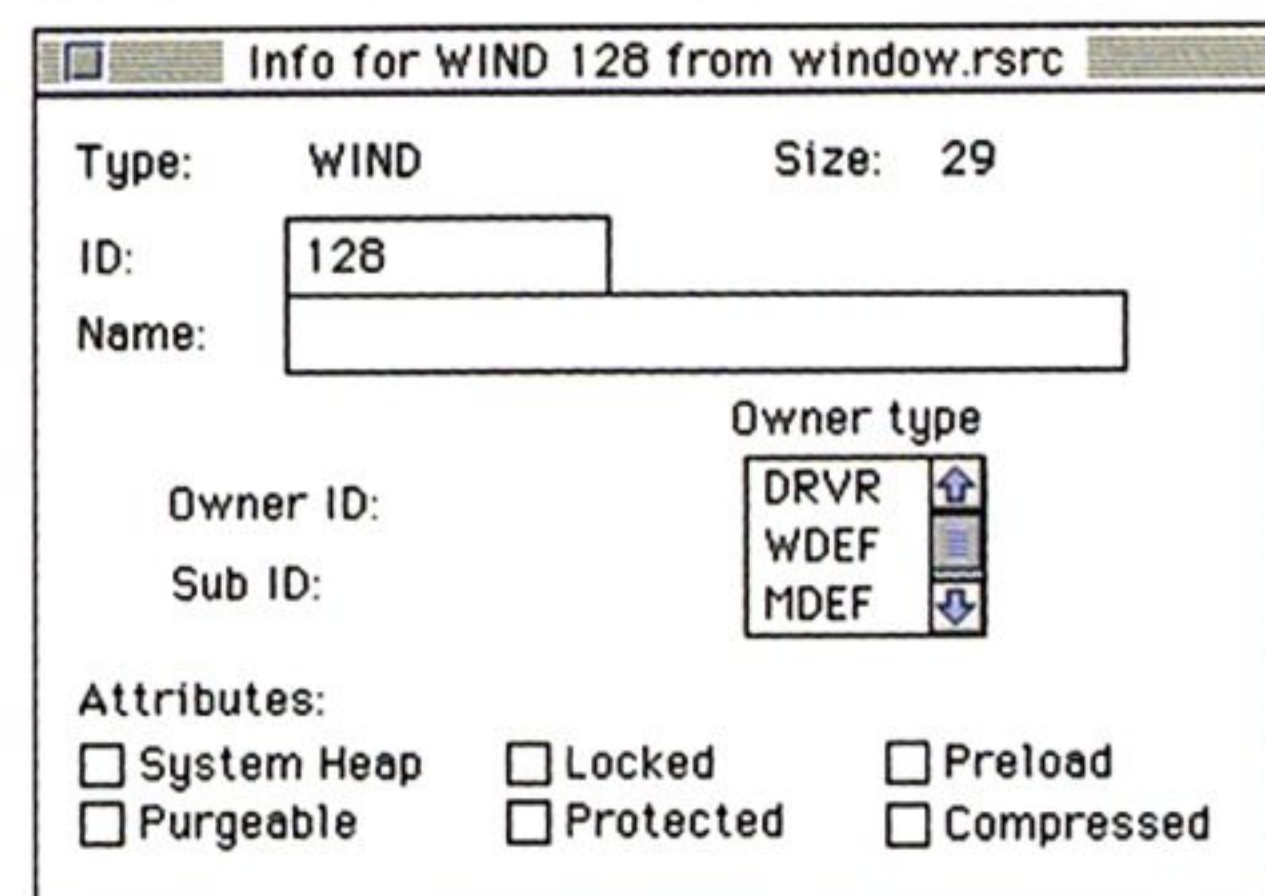
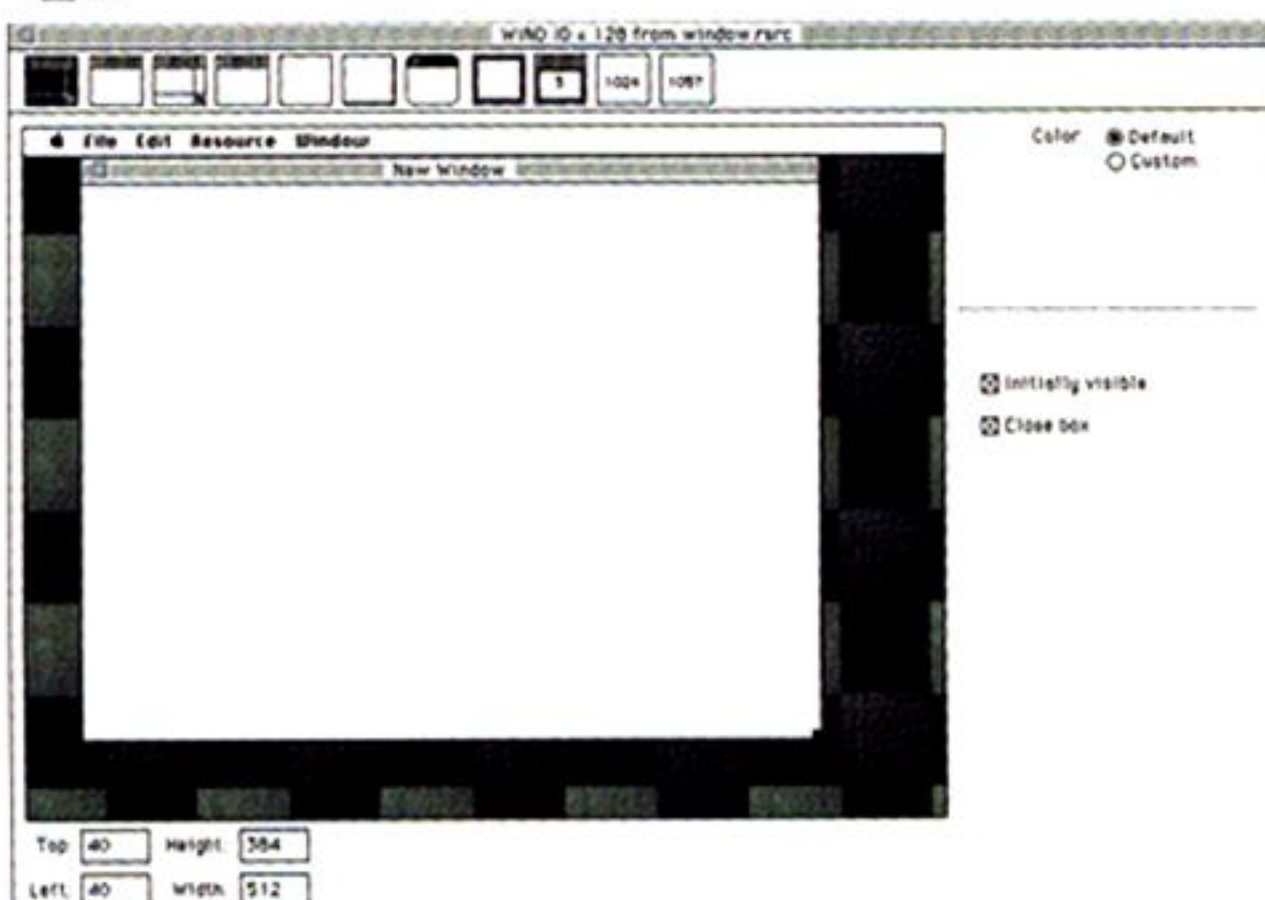
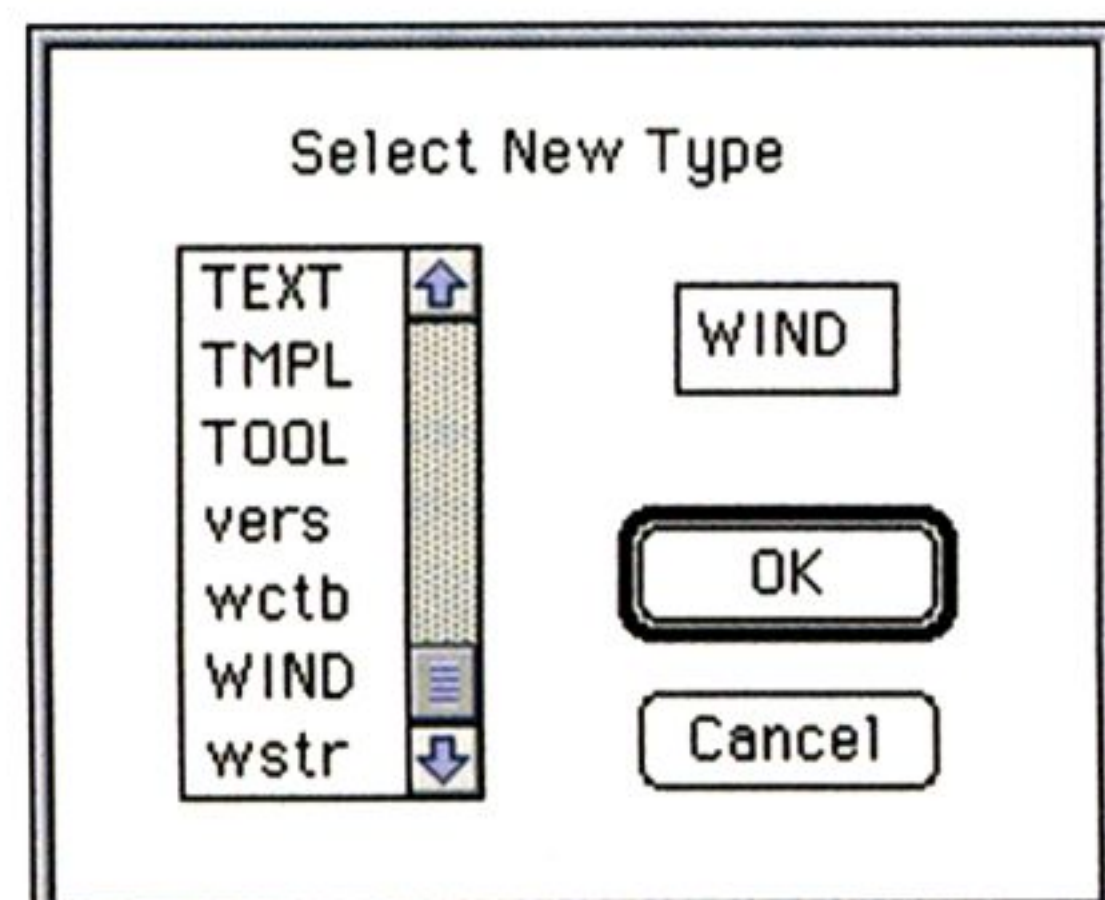
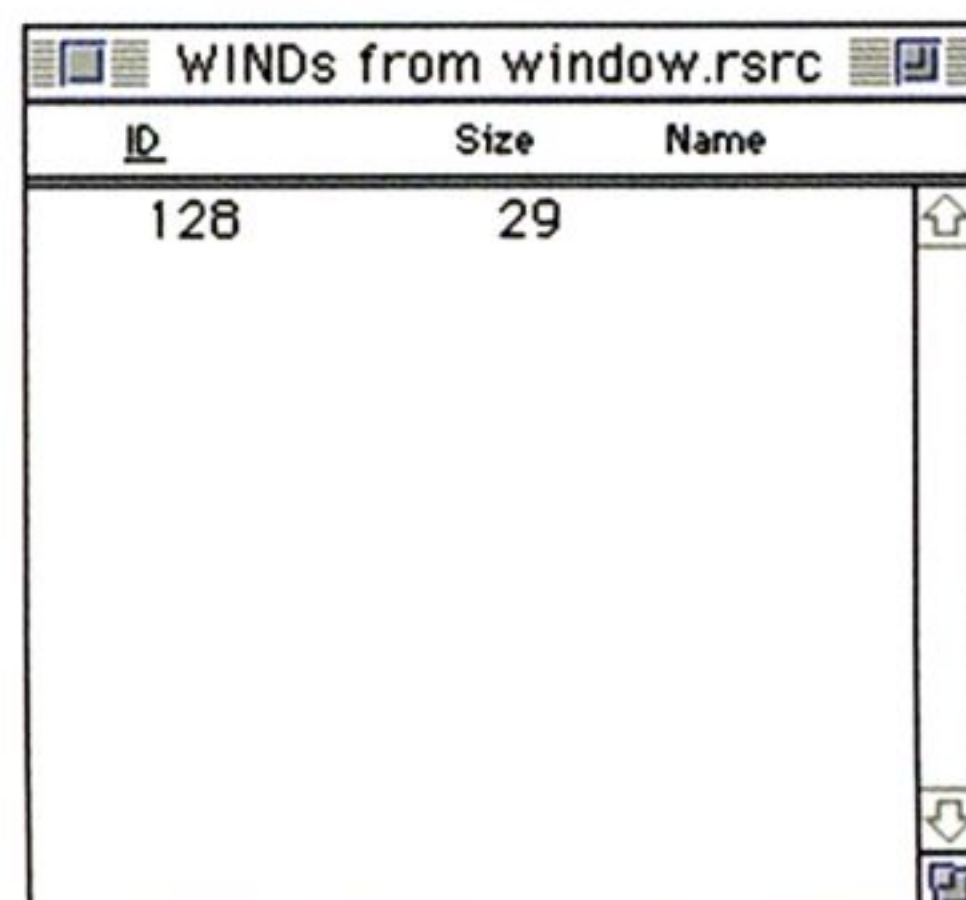



図10a

図10b

て開く必要があります。

2）WINDリソースをいじって遊ぶ

さっそくWINDエディタを見てみましょう。ウィンドウの左下の部分に，Top，Height，Left，Widthの4つのフィードがあります。前節では，このHeightとWidthの2つのフィールドに数値を入れたわけですが，今度はこの4つのフィールドにいろいろ数値を入れて，WINDエディタ上のプレビュー画面がどう変化するか試してみましょう。

WINDエディタの右下にある2つのチェックボックスはInitially visibleが，ウィンドウがこのリソースをもとにして作成されたとき，最初から表示するかどうかを選ぶもの，Close Boxが，ウィンドウのタイトル部分にあるクローズボックスを表示するかしないかを選ぶものです。Initially visibleはプレビュー画面では反映されませんが，このチェックボックスを外してアプリケーションを作成すると，起動したときにウィンドウが見えなくなってしまいます。

また，WINDエディタの上段にあるアイコンは，ウィンドウの形式を選ぶものです。これもいろいろ変えて試してみましょう。アイコン列の右側にある「？」マークのアイコンは，カスタムウィンドウのために用意されたものです。これをクリックすると，数値の入力を求めてくるダイアログが表示されますが，変更せずcancelをクリックしてダイアログを閉じます。

右上にあるColorラジオスイッチは変更せず，Defaultのままにしておきます。

ではこの4つフィールドに入力した数値や，選んだウィンドウの形状が本当にアプリケーションに反映されるかどうか確かめてみましょう。

それぞれのフィールドに適当な数値[※11]を入力して，window.rsrcを保存します。

次にCodeWarrior IDEに戻って，今度は実行/デバッグボタンをクリックし，コンパイルからアプリケーションの実行まで一気に行います。実行されたアプリケーションで表示されたウィンドウが，WINDエディタで入力した数値の位置，大きさになっているか，選んだウィンドウのかたちになっているか確かめてみてください。

最後にWINDリソースの数値を元に戻して保存することを忘れずに。

3）ウィンドウの名称と自動配置

今度はいまのところNewWindowとなっているウィンドウの名称を変えてみましょう。

ResEditに戻って，再びWINDエディタを開きます。WINDメニューからSet 'WIND' Characteristicsを選びます。ここで現れるのが図12のダイアログボックスです。さっそくWindow titleフィールドに適当な言葉を入れてみましょう。漢字など全角文字も受け付けてくれます。ここでは，昔懐かしいoh!mzと入れてみました。ダイアログを閉じると，すぐにプレビュー画面上のウィンドウのタイトルが変更されます。

ダイアログの下の2つのフィールド，refConとProcIDは，今回は変更しないでおきます。特にProcIDはWINDエディタで選択したウィンドウの形状などによって，それぞれ特定の値を取りますので，図12と違うからとあわてないようにしてください。

ウィンドウの名称変更も，CodeWarrior IDEに戻ってアプリケーションを作成し直せば即座に反映されます。

次はウィンドウの自動配置です。

これは，ユーザーごとに違う解像度のモニタを使っていても，アプリケーションを起動したときに，モニタ上のだいたい同じ位置にウィンドウを自動的に配置してくれるという便利な機能です[※12]。

使い方は簡単，WINDメニューからAuto Positionを選ぶだけです。これで図13のようなダイアログボックスが表示されます。最初はウィンドウ自動配置が無効になっていますから，左のポップアップメニューはnoneが選択され，左のポップアップメニューはグレイアウト（無効化）しています。ここで左のポップアップをCenterにしてみます。右のポップアップはMain Screenにしましょう。これでOKをクリック，リソース全体を保存します。アプリケーションを作成し直し，実行してみると……。

どうでしょう。ちゃんとスクリーンの真ん中にウィンドウが現れたと思います。解像度を変更しても，ちゃんと画面の真ん中に表示されますので試してみてください。

左のポップアップメニューには，ほかにAlert PositoinとStaggerがありますが，それぞれ画面中央の少し上くらいと画面左上端にウィンドウが表示されます。

これでResEditを使ってのWINDリソースの操作に関してのだいたいのところと，それがどうアプリケーションに反映されるかがなんとなくわかってきたのではないでしょうか。次の節はいよいよキャラの表示にチャレンジします。丹田に気合を込めつつ，ちんたら行ってみましょう。

※11　WINDエディタは，負の数も受け付けてくれますが，あんまり変な値は入れないようにしましょう。ResEdit自体がクラッシュしたり，window.rsrcが原因になってアプリケーションになにか問題を引き起こすかもしれません。

※12　これを知るまでのあいだ，筆者はいま使用されているモニタの表示領域と表示しようとしているウィンドウのサイズをもとに計算を行い，ウィンドウを移動させ，それから表示するという非常に手間のかかる作業を行わなければなりませんでした……。

嗚呼，憧れの画像表示

ついにやってきました，前半の山場です。

ウィンドウに絵を表示させるための下準備として，絶対にやらなければならないことが，ひとつあります……それは絵を用意することです。

PhotoshopやPainterで描いても，Shadeなどの3Dツールでレンダリングしても，写真をスキャニングしたり，デジカメで取り込んでもかまいません。とにかく，表示するためには絵が必要です[※13]。今回は，絵を用意できない方のためにimage01.pictという名前のPICTファイルを準備しましたが，もちろん自作のCGでも，お気に入りのCGでもかまいません。ただし，絵のサイズは，前節で設定したウィンドウの大きさに収めたほうが無難です（多少大きくても問題はありません）。

リストはdrawPict.cです。使う関数や変数型の数も増えてきましたが，順を追って覚えておけば，ほとんどはコピー&ペーストで用が足りる程度の定型句です。そのことを頭のどこかに引っ掛けつつ，つらつらと行ってみましょう。

1）プロジェクトの作成

またこれです。プロジェクト名はdrawPict.prj，ソース名はdrawPict.cとしておきます。ソースはちゃんとプロジェクトに追加し，いらないファイルは削除しておきましょう。

2）リソースの作成

PICTリソースを作成するには，ResEditのほかに画像をコピー&ペーストできるソフトが必要になりますが，市販/フリーを問わず，たいていの画像ソフトが対応しているので問題ないと思います。

まずResEditを立ち上げ，空のリソースファイルを作ります。これは，前に説明したとおり，プロジェクトがあるフォルダ内に作っておきましょう。

次に前節で使ったwindow.rsrcリソースファイルを開き，WINDリソースをdrawPict.rsrcにコピー&ペーストします。まず，wind.rsrcのリソースリストからWINDを選択・コピーし，drawPict.rsrcのリソースリスト

'WIND' Characteristics
Window title: oh! mz
refCon: 0
ProcID: 0
Cancel　OK

図12

Automatically Position the Window
(Works only with System 7.0 or later.)
Center　on　Main Screen
Cancel　OK

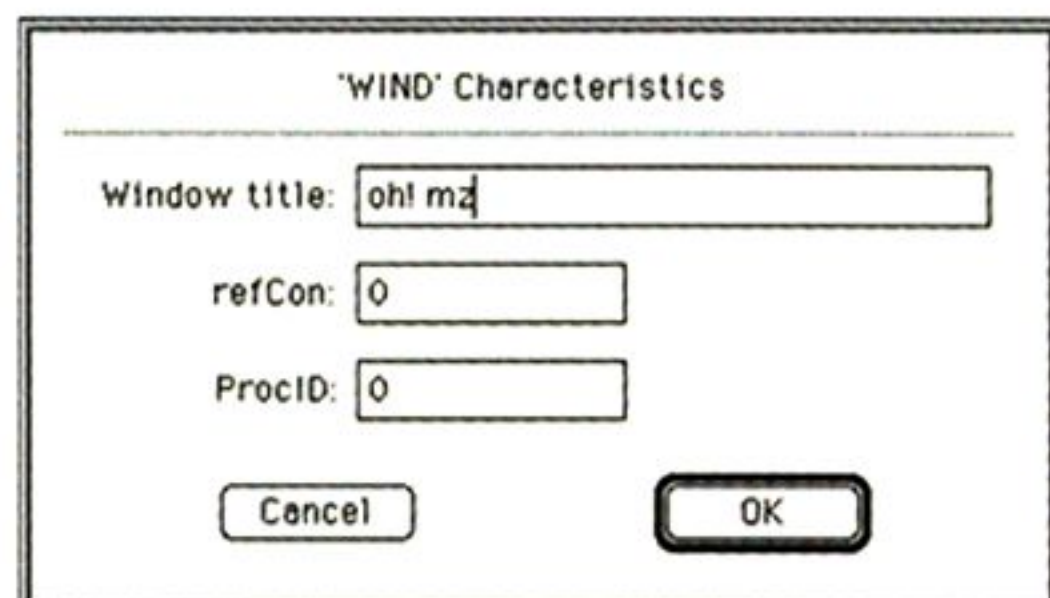
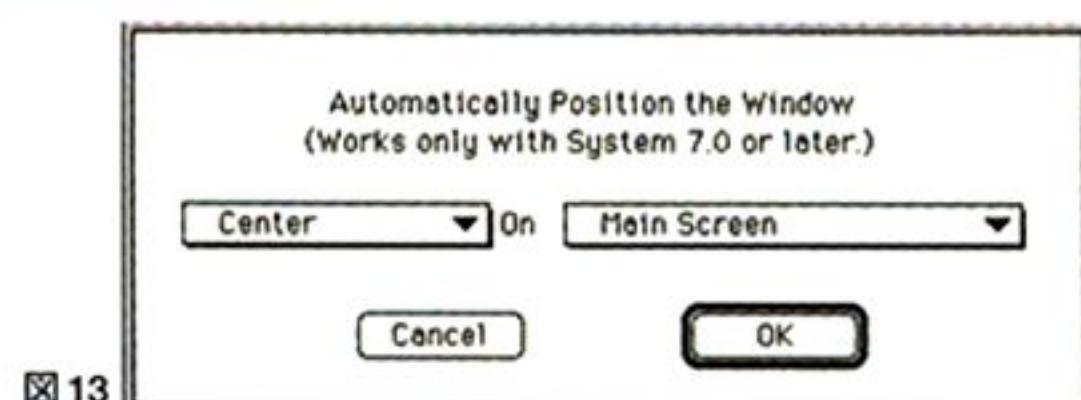



図13

にペーストすれば問題なく行えます。今回以降は，仕上がりの見栄えを意識してウィンドウの自動配置を使いますので，WINDリソースにAuto Positionが設定されていることを確認してください。左ポップアップメニューがCenter，右ポップアップメニューがMainScreenです。

確認したらWINDリソースを閉じましょう。

さて，今度はPICTリソースの番です。drawPict.rsrcのリソースリストがいちばん手前にきている状態で，WINDリソースを作ったときと同じように新規リソースを作成します。ここでまた図9のようなダイアログが出てきますが，今度はPICTを選びましょう。すると，すぐにPICTリソースリストとPICTプレビュー用のウィンドウが開きます。PICTリソースは作られたばかりで空の状態ですから，開いたリソースリストとプレビュー用ウィンドウは両方とも空白のままです。

ここで画像ソフトを立ち上げ，準備しておいた画像か自前の絵がない人はimage01.pictを読み込んでコピーします。ResEditに戻ったらPICTリソースリストか，PICTプレビュー用ウィンドウでペーストしましょう。このとき，もし図14のようなダイアログが出てきたらYesをクリックしてください。

このダイアログは，ペーストするリソース（この場合は画像）に割り振られているID番号が，元からあるリソースの番号と同じ場合に上書きを許可するかどうか聞いてきているもので，Yesで上書きの許可，Noでペーストの中止，Unique IDで重ならないようなID番号に変更してペーストを行うようになっています。

無事画像をペーストできたら，PICTリソースリストに戻り，いまペーストした画像のID番号を200にします。WINDリソースを作ったときと同じように，Resourceメニューから Get Resource Infoを選ぶかCommand + Iで情報ウィンドウを開き，ID番号を変更しましょう。変更し終わったあとのPICTリソースリストは図15のようになっているはずです（画像は，ペーストした内容によって異なります）。

これでdrawPictで使用するリソースが完成しました。

drawPict.rsrcファイルを保存し，ResEditを終了させましょう。

3）リソースの追加からアプリケーションの実行まで

Smallest.prjやwindow.prjのときと同じように，プロジェクトにいま作ったリソースを追加し，元からあるリソースファイルを削除します。

あとはコンパイルして実行するだけです……。

どうでしょう？　ちゃんと図16のようにウィンドウにPICTが表示されたでしょうか？

表3　ああ，憧れのキャラ表示

```
/*QuickDrawが現在絵を描くための対象としているグラフポートをもらう*/
void GetPort( GrafPtr *port );
引数：*port 現在のグラフポートへのポインタ
返り値：なし，ただし，関数から戻ってきたときに*portには現在の描画対象グラフポート（カレントポート）が入っている

/*QuickDrawに絵を描くグラフポートを指定する*/
void SetPort( GrafPtr port );
引数：port  絵を描かせたい（カレントポートにしたい）グラフポート
返り値：なし


/*PICTリソースから絵を取り出す*/
PicHandle GetPicture( short pictureID );
引数：pictureID ほしいPictリソースのID
返り値：Pictureへのハンドル。指定されたIDにPICTリソースがなかった場合はNILになる


/*ハンドルをロックする*/
void Hlock( Handle theHandle );
引数：theHandle ロックしたいハンドル
返り値：なし


/*ハンドルに対するロックを解除する*/
void Hlock( Handle theHandle );
引数：theHandle ロックを解除したいハンドル
返り値：なし


/*絵を描く*/
void DrawPicture( PicHandle thePict , const Rect *drawRect );
引数thePict：描画したいPictureへのハンドル
     drawRect：描画したい領域
返り値：なし


/*リソースを捨てて，メモリを解放する*/
void ReleaseResource( Handle theResource);
引数：捨てたいリソースへのハンドル
返り値：なし
```

コンパイル時にエラーが出ないのに，ウィンドウが出てこなかったり，PICTが表示されなかったりした場合は，リソースのID番号をよく確かめてみてください。また，絵が大きすぎる場合も表示されないことがあります。

4）drawPict.cの中身を見る

ウィンドウに画像表示できるようになったからといって，実はそんなに長くないdrawPict.cなのですが，画像を表示するためにどんなことをしているのか追ってみましょう。

initToolBox ()関数やGetNewCWindow ()関数などの部分は前と同じなので，飛ばして目新しいところを見ていきましょう。

GetPort ()関数は，呼び出された時点でQuickDrawが描画対象としているウィンドウ，カレントポートを得るために使われます。カレントポートはmain ()関数の最初のほうで宣言されているWindowPtr型の変数，oldWindowPtrに保存されます。

次は，SetPort ()関数です。これは，引数として与えられたWindowPtrをカレントポートにします。ここでは，直前にGetNewCWindow ()関数で作ったウィンドウ，つまりPICTリソースの絵を描画するために作ったウィンドウをカレントポートにします。これで描画のためのウィンドウの用意ができました。drawPict ()関数でいよいよウィンドウへの描画をします。

リスト3　drawPict.c

```
#define kWindow_ID  200
#define kIN_FRONT (WindowPtr) -1
#define NIL 0

#define kPict_ID 200

/* 関数プロトタイプ */
void main(void);                                                    /*メイン関数*/
void initToolBox(void); /*ToolBox初期化*/
void drawPict(int);/* ウィンドウにPictリソースの絵を表示する*/

/* グローバル変数 */

/* メイン関数 */
void main(void)
{
   WindowPtr mainWindowPtr;/*Window構造体を保存するためのポインタ*/
   WindowPtr oldWindowPtr;/*Window構造体を避難させるためのポインタ*/

   initToolBox();/*ToolBox初期化*/

   /*リソースの情報をもとにウィンドウを作る*/
   mainWindowPtr = GetNewCWindow( kWindow_ID, NIL , kIN_FRONT );
   if( mainWindowPtr == NIL )
   {
      /*失敗したら、なにもせずにFinderへ戻る*/
   ExitToShell();
   }

   GetPort(&oldWindowPtr);/* 現在の描画用ウィンドウのポインタを保存する*/
   SetPort(mainWindowPtr);/* 描画用ウィンドウをいま生成したウィンドウに設定する。*/

   drawPict( kPict_ID );/* リソースの中のPictをウィンドウに表示する */

   SetPort(oldWindowPtr);/* 描画用ウィンドウをもとに戻す。*/

   SysBeep(50);

   while (!Button()){};/* マウスボタンが押されるまで待つ*/

   return;
   /*終わり*/
}

/* 初期化 */
void initToolBox()
{
   /*必要な各種マネージャの初期化。*/
   InitGraf(&qd.thePort);
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs(nil);
   InitCursor();
}

/* リソースに入っているPictを表示する*/
void drawPict(int pictID)
{
   PicHandle thePictHandle;
   /*Pictを保持するためのハンドル*/

   thePictHandle= GetPicture( pictID );
   /*Pictリソースを読み込む*/
   if(thePictHandle == NIL){ return; }
   /*リソースが無かったりしたら、そのまま帰る*/

   HLock( (Handle)thePictHandle);
   /*  Resource操作中にちょっかいを出されるのを防ぐ*/

   DrawPicture( thePictHandle , &((*thePictHandle)->picFrame));/* 絵を描く */

   HUnlock( (Handle)thePictHandle);
   /*  ロック解除*/
   ReleaseResource( (Handle)thePictHandle);/* メモリを占有している絵（PICT）を破棄する。*/
}
```

この関数は，カレントポートに引数として渡されたIDを持つPICTリソースを描画する関数で，詳しいことはあとで説明します。

描画が終わったらoldWindowPtrに保存していた元のカレントポートをSetPort ()関数で元に戻して後始末をします。

つまり，ウィンドウへの描画を行う場合，実際に描画を行うための部分とSetPort ()，GetPort ()関数の関係は以下のようになります。

```
GetPort（＆カレントポート退避用変数）；
SetPort（描画させたいウィンドウ）；

/＊実際に描画を行う部分＊/

SetPort（カレントポート退避用変数）；
```

これはもう，ウィンドウへの描画を行うときの公式みたいなもので，ちゃんと覚えておけばあとあと便利な部分です。あとで出てくる仮想画面への描画を行うときにも，関数名は異なりますが，同じ形の公式が出てきます。複数のウィンドウを扱うプログラムを作ることになったとき，この退避・設定・描画・復旧はさらに大切なものになってくるはずです。

では，main ()関数から呼び出されているdrawPict ()関数を見てみましょう。この関数はウィンドウにリソースから取り出した絵を表示するもので，drawPicture.prjの心臓部です。

GetPicture ()関数は，与えられた引数と同じIDのPICTリソースを持ってくる関数です。返り値はPicHandle型の変数に収められます。PicHandle型は，ToolBoxがPICTを扱うために使う変数型でリソースから絵を持ってきたり，ファイルから読み込んだりするときにお世話になります。引数で渡したIDにPICTリソースがなかったり，PICTリソースが大きすぎてメモリ不足になった場合はGetPicture ()はNILを返します。そのときは描画を行わずdrawPict ()関数から抜けることになります。

HLock ()関数は引数として渡されたハンドルをロックします。

ハンドルはC言語でいうところの「ポインタのポインタ」で，マスタポインタを介して実際に割り当てられたメモリにアクセスすることができます。ハンドルとメモリを仲介するマスタポインタはメモリマネージャによって管理されており，アプリケーションが使わないメモリ領域を整理する（ヒープの圧縮）ときなどに変更されることがあります。普通はすぐにその後始末をしてくれるのですが，大きなデータを扱うときなどはそれが間にあわなかったりする可能性もあります。そのため，プログラムがメモリに対して作業しているあいだにメモリマネージャにちょっかいを出されたくない場合は，「作業中！　いじるな！」の看板を出しておく必要があります。この看板を出してくれるのがHLock ()関数なのです。HLock ()関数は，ハンドルであれば，なんでもロックしてくれますが，PicHandle型などはHandleにキャストする必要があります。

描画などでPICTリソースを扱う場合，以下のように実際にリソースを扱う部分をHLock ()関数，HUnlock ()関数ではさんでおけば安心して扱うことができます。

```
HLock（（Handle）リソースから取り出したPICT）；

/＊PICTを使う＊/

HUnlock（（Handle）リソースから取り出したPICT）；
```

HUnlock ()関数は，HLock ()関数と反対の働きをする関数で，ハンドルにかけられたロックを解除します。ロックしたハンドルをそのまま放置すると，プログラムが大きくなってきたときいろいろ面倒なことが起こるので忘れないように。

最後にHLock ()とHUnlock ()のあいだにはさまれたDrawPicture ()関数を見てみましょう。DrawPicture ()は，PICTをカレントポートに描画する関数で，2つの引数を持っています。

最初の引数は，描いてほしいPICTをPicHandle型の変数で渡します。これは面倒くさいことは考えずに，GetPicture ()で手に入れたPicHandle型変数，ここではthePictHandleをそのまま渡せば大丈夫です。

2つ目の引数は，ひとつ目の引数で渡したPICTをカレントポートのどこに描くかを指定します。この指定には，Rect型を使います。Rectは画面上で長方形の形と位置を指定するためのもので，ここではthePictHandleの中に入っている絵のサイズをそのまま渡すことにしましょう。ここで示しているPICTのサイズを取り出す方法は，覚えておくとなにかと便利です[※14]。

最後のReleaseResource ()関数は，GetPicture ()関数で取り出したPICTリソースを破棄しメモリの掃除をしてくれます。複数のリソースを扱うようなアプリケーションを作るようになってくると，使わなくなったリソースがメモリの中に段々と積み上がってきてメモリを圧迫し，問題を起こすようになってきます。必要な絵のリソースを使い終わったら，ReleaseResource ()関数を呼び出して掃除をするクセをつけておきましょう。

これでようやくdrawPicture関数の説明が終わりました。

drawPict.cはウィンドウを開き，PICTリソースから絵を取り出して開いたウィンドウに描画するというかなり単純なプログラムではありますが，なかにはこれからゲームを作るために必要なことがたくさん含まれています。この説明とリストの内容と突きあわせながら，じっくりと調べてみてください。

ようやっとウィンドウに絵を表示できました。全体の半分はクリアできたと考えてもよいでしょう。

※13　なぜこれほど絵にこだわるかというと，ゲームを作るにあたって絵を準備することは，プログラムをするのと同じくらい大変なことだからです……ホント大変なのです…もちろん，音楽も……。

※14　PictHandle型の構造体のなかには絵の大きさを示すRect型も含まれています。

Press the Unique ID button if you want new resource IDs assigned to the resources with conflicting IDs.

Replace resources with the same ID?

Unique ID　No　Yes

図14

PICTs from drawPict.rsrc

200

図15

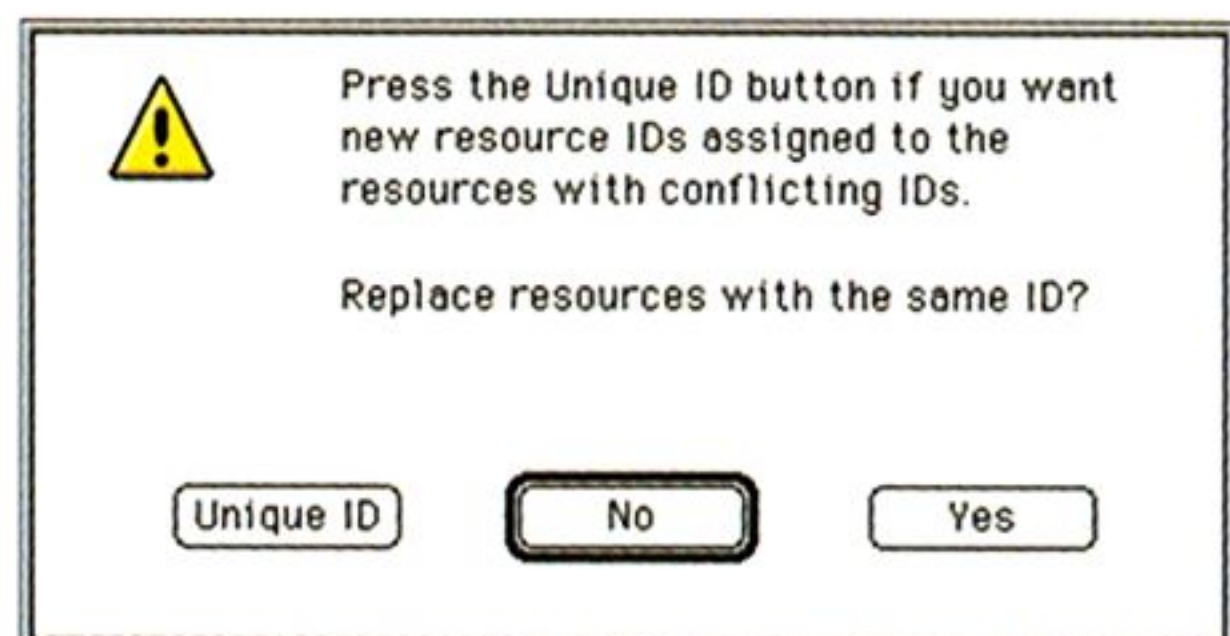
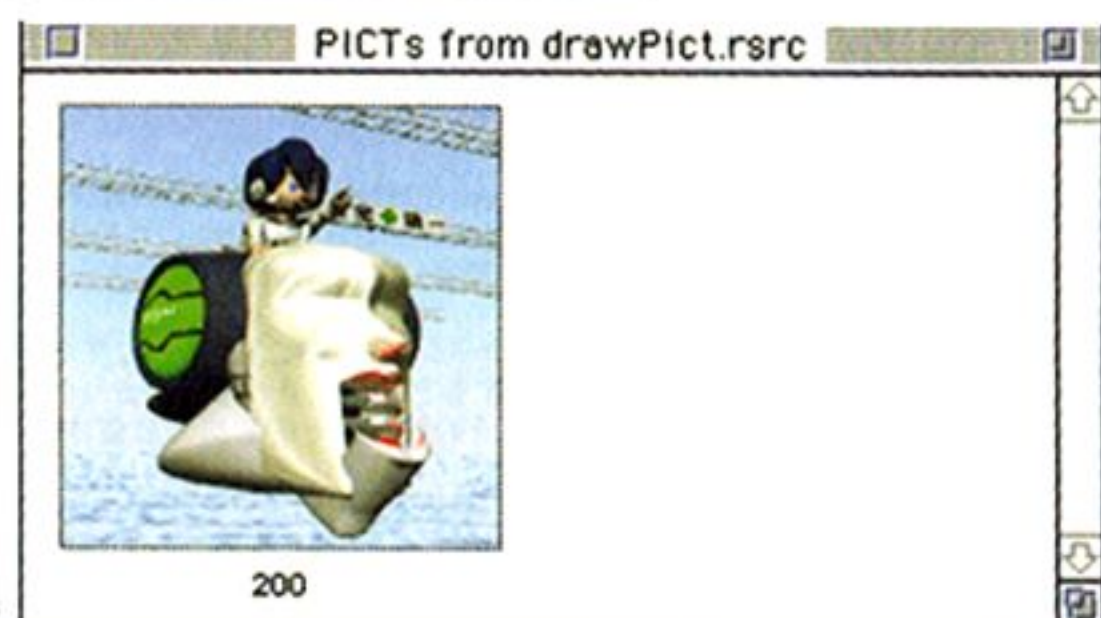
動く標的

この節のお題目はもちろん「キャラクターを動かしてみること」です。ウィンドウ上でキャラクターを動かすことができるようになれば，この記事の目標であるシューティングゲームの作成も夢ではなくなってきます。今回はオフスクリーンGWorldという仮想画面の説明もあり，てんこもりです。さっそく行ってみましょう。

1）アプリケーションの作成と実行

プロジェクト名はmovePict.prj，ソース名はmovePict.cで行きましょう。ここらへんは毎度の作業です。

リソースは前節のdrawPict.rsrcをコピーして使うことにしましょう。ファイル名はmovePict.rsrcに変更します。また，PICTリソースは実際にシューティングゲームで使えるものにしましょう。サイズは64×64ドットとします。前節と同じように，絵が準備できない方のためにship01.pictを用意しました。これを，PICTリソースにペーストします。手順は前節のままです。IDは200のままにしておきましょう。

例によって，プロジェクトに必要なファイルを追加し，コンパイルして実

行です。

2）動かない標的

アプリケーションを立ち上げた途端，画面中央にウィンドウが開きかけて，そのまま終了してしまったと思います……なぜなんでしょう？

movePict.cは，内部でなんらかの問題が起こったらなんの警告も出さずExitToShell()で突然終了するようになっています[※15]。しかし，ウィンドウが開きかけたことから，GetNewCWindow()での問題ではなく次にExitToShell()があるところ＝オフスクリーンGWorldを作るための自前の関数，initOffScreen()のなかで問題が起きているのは間違いありません。どこで問題が起きているのかはわかりましたが，いったいなにが起きているのでしょう？

Macのアプリケーションが不具合を起こしたときの経験的対処法のなかに「割り当てメモリを増やす」というのがあります。そこで，このアプリケーションに対する割り当てメモリも増やしてみることにしましょう……だいたい4096Kバイトくらいにしておきます（もっと少なくても動作するでしょう）[※16]。

どうでしょう，今度は無事起動して，ウィンドウの中央をキャラクターが左右に動いているはずです。

結局，メモリが足りなかったのです。割り当てメモリさえ増やせばちゃんと動くことはわかりましたが，アプリケーションを作ったときいちいち割り当てメモリを増やさなければならないのは面倒です。そこで，CodeWarrior IDEに戻って割り当てメモリの設定をすることにしましょう。これには，ツールバーかプロジェクトウィンドウにある図17のボタン（ターゲット設定ボタン）を使います。

ボタンを押すと，図18のようなターゲット設定パネルが現れます。ここ

リスト4　movePict.c

```
#define kWindow_ID  200
#define kIN_FRONT  (WindowPtr)-1
#define NIL  0

#define kPict_ID  200

/* 関数プロトタイプ */
void main(void);/* メイン関数 */
void initToolBox(void);/*ToolBox初期化 */
void drawPict(int);/* カレントポートにPictリソースの絵を表示する */

void initOffScreen(GWorldPtr *,Rect *,int);/* オフスクリーンGWorldの作成 */

void drawObject(int,int);/* 裏画面にキャラを描く */
void updateScreen(void);/* 裏画面を表画面に転送する。裏画面は塗りつぶして掃除する */

/* グローバル変数 */
WindowPtr mainWindowPtr;/*Window構造体を保存するためのポインタ */
GWorldPtr offScreenPtr;/* 仮想画面用オフスクリーンGWorld*/
GWorldPtr cgPatternPtr;/* キャラ保存用オフスクリーンGWorld*/

Rect mainWindowRect;/* ウィンドウの大きさ */

/* メイン関数 */
void main(void)
{
  int  h0;/* 座標用 */
  int dir;/* 方向 */

  initToolBox();/*ToolBox初期化 */

  /* リソースの情報をもとにウィンドウを作る */
  mainWindowPtr = GetNewCWindow( kWindow_ID, NIL , kIN_FRONT );
  if( mainWindowPtr == NIL )
  {
    /* 失敗したら、なにもせずにFinderへ戻る
  */ExitToShell();
  }

  /* ウィンドウの大きさを取り出しやすくするために、グローバル変数に入れておく */
  mainWindowRect = mainWindowPtr->portRect;

  /* オフスクリーンGWorldを作る */
  initOffScreen(&offScreenPtr, &mainWindowRect , -1);

  /* キャラ用のGWorldを作る */
  initOffScreen(&cgPatternPtr, &mainWindowRect ,kPict_ID);

  SysBeep(50);

  /* 座標初期化 */
  h0 = 262;
  dir = -2;

  /* マウスボタンが押されるまでループ */
  while (!Button())
  {
    drawObject( h0 , 160);

    h0 += dir;
    if((h0 < 16 )||( 432 < h0 )){ dir = -dir;}
    updateScreen();
};

  return;
  /* 終わり */
}

/* 初期化 */
void initToolBox()
{
  /* 意地でもアプリケーションヒープを最大にする */
  MaxApplZone();

  /* 必要な各種マネージャの初期化。 */
  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(nil);
  InitCursor();
}

/* リソースに入っているPictを表示する */
void drawPict(int pictID)
{
  PicHandle thePictHandle;/*Pictを保持するためのハンドル */

  thePictHandle= GetPicture( pictID );/*Pictリソースを読み込む */

  if(thePictHandle == NIL){ return; }/* リソースが無かったりしたら、そのまま帰る */

  HLock( (Handle)thePictHandle);
  /*  Resource操作中にちょっかいを出されるのを防ぐ */

  DrawPicture( thePictHandle , &((*thePictHandle)->picFrame));/* 絵を描く */

  HUnlock( (Handle)thePictHandle);
  /*  ロック解除 */
  ReleaseResource( (Handle)thePictHandle);/* メモリを占有している絵（PICT）を破棄する。 */
}

/* オフスクリーンGWorldを作る */
void initOffScreen(GWorldPtr *theGWorldPtr,Rect *theRect,int pictID)
{
  //一時保存用
  GWorldPtr saveGWorldPtr;
  DHandle saveGDHandle;

  //  BG用オフスクリーン（NewGWorld）確保できなかった場合は、終了する。
  *theGWorldPtr = NIL;
  if( NewGWorld( theGWorldPtr , NIL , theRect , NIL , NIL , NIL ) != noErr)
  {
  ExitToShell();/* 確保できなかったら、終了する */
  }

  if(pictID == -1){return;}/*pictIDに-1が指定されていた場合、PICTの描画はしないでそのまま帰る */

  GetGWorld( &saveGWorldPtr , &saveGDHandle );/* もとのGWorld保存（作ったGWorldに画像をロードするため） */
  LockPixels( GetGWorldPixMap( *theGWorldPtr ));/* 描画のためのピクセルロック */
  SetGWorld( *theGWorldPtr , NIL );/* 確保したGWorldを描画ポートに設定する */

  drawPict(pictID);/*PICTの描画 */

  UnlockPixels( GetGWorldPixMap( *theGWorldPtr ));/* ピクセルロック解除 */
  SetGWorld( saveGWorldPtr , saveGDHandle );/* 描画ポート復旧 */
}

/* 裏画面にキャラを描く */
void drawObject(int x0,int y0)
{
  Rect srcRect,dstRect;/* 描画範囲を保持する変数。 */

  SetRect(&srcRect,0,0,64,64);/* 領域を決める。
  */dstRect = srcRect;

  OffsetRect( &dstRect , x0 , y0 );/* 実際に書き込む位置にRectをオフセットする。 */

  CopyBits(&((GrafPtr)cgPatternPtr)->portBits,/* 転送もとGWorldまたはWindowPtr*/
    &((GrafPtr)offScreenPtr)->portBits,/* 転送先GWorldまたはWindowPtr*/
    &srcRect,
    /* 転送元Rect*/
    &dstRect,
    /* 転送先Rect*/
    srcCopy,
    /* 転送方法(この場合はベタ転送)*/
    NIL);
    /* マスク用Region、マスクしないのでNIL*/
}

/* 裏画面を表画面に転送する。裏画面は塗りつぶして掃除する */
void updateScreen(void)
{
//一時保存用
GWorldPtr saveGWorldPtr;
GDHandle saveGDHandle;

  //  メインスクリーンに絵を転送する
  CopyBits(&((GrafPtr)offScreenPtr)->portBits,/* 転送元GWorldまたはWindowPtr*/

    &((GrafPtr)mainWindowPtr)->portBits,/* 転送先GWorldまたはWindowPtr*/
    &mainWindowRect,/* 転送元Rect*/
    &mainWindowRect,/* 転送先Rect*/
    srcCopy,/* 転送方法(ベタ転送)*/
    NIL);
    /* マスク用Region、マスクしないのでNIL*/

  GetGWorld( &saveGWorldPtr , &saveGDHandle );/* 元のGWorld保存 */
  LockPixels( GetGWorldPixMap( offScreenPtr ));/* 描画のためのピクセルロック */
  SetGWorld( offScreenPtr , NIL );/* 確保したGWorldを描画ポートに設定する */

  EraseRect(&mainWindowRect);//ウィンドウ内を塗りつぶしておく。
  /* このEraseRectをコメントアウトすると初期化されたばっかりのGWorldがどうなっているか見れます */
  /* 適当なアドレスが割り振られて、その状態がそのままで放置されているので結構愉快です。 */

  UnlockPixels( GetGWorldPixMap( offScreenPtr ));/* ピクセルロック解除 */
  SetGWorld( saveGWorldPtr , saveGDHandle );/* 描画ポート復旧 */
}
```

ではアプリケーションに関わるいろいろな設定を行うことができますが，今回はとりあえず右側に並んでいるフィールドのうち，推奨ヒープサイズと最小ヒープサイズを4096Kとします。また，ファイル名の部分を「movePict DEBUG PPC」と変更してみましょう。変更を終えてパネルを閉じるか，右下の保存ボタンを押すと，再度リンクかコンパイルする必要があると警告するダイアログが出てきますが，これはそのままOKとします。再度アプリケーションを作ってみましょう。

すると今度は，先ほどターゲット設定パネルのファイル名の部分で指定した名前，この場合は「movePict DEBUG PPC」というアプリケーションが現れます。このアプリケーションの情報を見て，割り当てメモリが最初から4096以上になっていることを確認しましょう[※17]。

3）movePict.cの中身を見る

いままでのソースに較べて，倍近いボリュームとなっていますが，半分はコメントです。順を追って見ていくことにしましょう。

まずいままでのソースと異なっているのは，いままでmain()関数の中にあったmainWindowPtrなどの変数がグローバル変数として定義されていることです。これは裏画面やメインウィンドウに，どの関数からでも描画できるようにするためです。また，メインウィンドウの大きさを保存しておくために，Rect型のグローバル変数，mainWindowRectも定義しています。

main()関数そのものの長さは，さほど変わっていません。

最初にきているのは表示するキャラクターを動かすための変数です。

ToolBox初期化のためのinitToolBox()からウィンドウの確保までは変更がありません。次に，よく使うメインウィンドウのサイズをグローバル変数に保存しています。

今回のミソその1であるinitOffScreen()関数は，仮想画面であるオフスクリーンGWorldを確保して，さらに必要であればそこにPICTリソースの絵まで表示してくれるというスグレものです。

裏画面用とキャラ用のオフスクリーンGWorldを作ったあとは，SysBeep()で音を鳴らし，キャラクターを動かすための座標と方向を設定，ループに入ります。

いままではなにもない空ループだったところに，キャラクターを描画するためのdrawObject()，裏画面を表画面であるメインウィンドウに転送するためのupdateScreen()が入っています。キャラを動かしている2行に関しては説明するほど大したモンじゃないので省略します。

メインはこれだけです。

仮想画面が使えるようになったおかげで直接メインスクリーンに描画する必要がなくなり，GetPort()，SetPort()が取れてスッキリしました。

次はinitToolBox()関数です。

ほとんど変わりありませんが，最初にMaxApplZone()関数が追加されています。これは，アプリケーションが割り当てメモリを全部使うと宣言するもので，プログラムの最初に呼び出す必要があります。これは，MacOSがアプリケーションに対してメモリを割り当てるとき，「情報を見る」で指定されている使用サイズまたは空きメモリの容量を上限として，必要に応じて順次割り当てるというかたちを取っているためです。

まあ，難しい説明はともかく「プログラムの最初で絶対に呼び出す関数」として，ToolBox初期化の前に呼び出しておけば，まず間違いないでしょう。

drawPict()関数は変更されていません。説明を忘れてしまった場合は前節を見てください。

さて，いよいよ今回のキモの部分。オフスクリーンGWorldの登場です。ゲームに限らず，これからアプリケーションを作るにあたって相当にお世話になるであろう関数が多数出てきますので，じっくりと追ってみてください。

まずはinitOffScreen()関数です。

最初に出てくるGWorldPtr型は，先に登場したWindowPtrと同じように，絵を描くためのスクリーンを扱うときによく使う構造体へのポインタです。WindowPtrと異なる点は画面に表示させないスクリーン，仮想画面を扱うという点にあります。もうひとつの，GDHandle型は，デバイスといわれるハードウェアに関わる部分を引き受けてくれるもので，このようなかたち以外で扱うことはほとんどありません（ドライバとか作る人は別かもしれませんが……）。必要な定型句だと理解してください。

さて，いよいよオフスクリーンGWorldを作ります。

NewGWorld()関数は，引数から新しいGWorldを作成する関数です。これ一発で新しいオフスクリーンGWorld，つまり仮想画面を作ってくれるという便利な関数です。引数が多く，少し腰が引けるかもしれませんが，実際に使うのは最初の3つくらいです。詳しい説明は関数表のほうを見てください。今回はまた色深度を指定する引数に0を指定したので，アプリケーションを動かすとき，モニタが何色に設定されているか気にする必要はあり

表4　動く標的

```
/*オフスクリーンGWorldを作成する*/
QDErr NewGWorld( GWorldPtr *theGWorld , shot colorDepth , const Rect *theRect , CTabHandle
colorTable, GDHandle theGDevice , GWorldFlags theFlag);
引数：theGWorld 戻ったとき，作成したGWorldへのポインタが入っている
    ：colorDepth 色深度を指定する。1ピクセルあたりのビット数で指定し，8なら256色，24だと1600
      万色になる。0だと現在のモニタの色深度を引き継ぐ。複数のモニタがある場合はそのなかでいちば
      ん色深度の高いものが選ばれる
    ：theRect 作成するGWorldの大きさを指定する。0だといちばん色深度の高いモニタの大きさをもら
      ってくる
    ：colorTable 設定するカラーテーブル。先にcolorDepthに0を指定していた場合，この値が無視され，
      選んだモニタのカラーテーブルをもらってくる
    ：theGDevice 描画環境で使用するデバイス。0を指定したほうが無難
    ：theFlag 設定フラグ。0を指定しておくのが無難
返り値：QuickDrawで発生したエラー値。noErr（これはシステムで定義された定数）でなかったらなにか
      エラーが起きている


/*QuickDrawが現在絵を描くための対象としているグラフポートをもらう*/
void GetGWorld( CGrafPtr *thePort , GDHandle *theGDHandle );
引数：thePort 戻ったとき，現在のGWorldへのポインタが入っている
    ：theGDHandle 戻ったとき，現在のデバイスへのハンドルが入っている
返り値：なし


/*PixMapの画像メモリをロックする*/
Boolean LockPixels( PixMapHandle theHandle );
引数：theHandle ロックするPixMapへのハンドル
返り値：ロックできなかったりしたらfalse


/*GWorldのPixMapへのハンドルをもらう*/
PixMapHandle GetGWorldPixMap( GWorldPtr theGWorld );
引数：theGWorld PixMapへのハンドルがほしいGWorld
返り値：PixMapへのハンドル


/*QuickDrawに絵を描くGWorldを指定する*/
void SetGWorld( CGrafPtr thePort , GDHandle theGDHandle );
引数：thePort絵を描かせたいGWorldへのポインタ
    ：theGDHandle 絵を描かせたいデバイスへのハンドル。通常は0を指定する
返り値：なし


/*Rect型に値を代入する*/
void SetRect( Rect *theRect , short left , short top , short right , short bottom );
引数theRect ：値を代入されるRect
    leftRect ：で示される長方形の左上のX座標
    top ：左上のY座標
    right ：右下のX座標
    bottom ：右下のY座標
返り値：なし


/*Rect型で示される領域を移動させる*/
void OffsetRect( Rect &theRect , short x0 , short y0 );
引数：theRect
移動させたいRect ：x0
X軸方向の移動量：y0
Y軸方向の移動量
返り値：なし


/*画像の転送*/
void CopyBits( const BitMap srcBits , const BitMap dstBits , const Rect srcRect , const Rect dstRect
, short mode , RgnHandle maskRegion );
引数：srcBits
転送元GWorldまたはWindowPtr ：dstBits
転送先GWorldまたはWindowPtr ：srcRect
転送元Rect ：dstRect
転送先Rect ：mode
転送方法,srcCopy（そのままコピー）かditherCopy（ディザ処理する）でほとんどは用が足りる
maskRegion ：マスク用Region。今回は使わないので0を指定する
返り値：なし
```

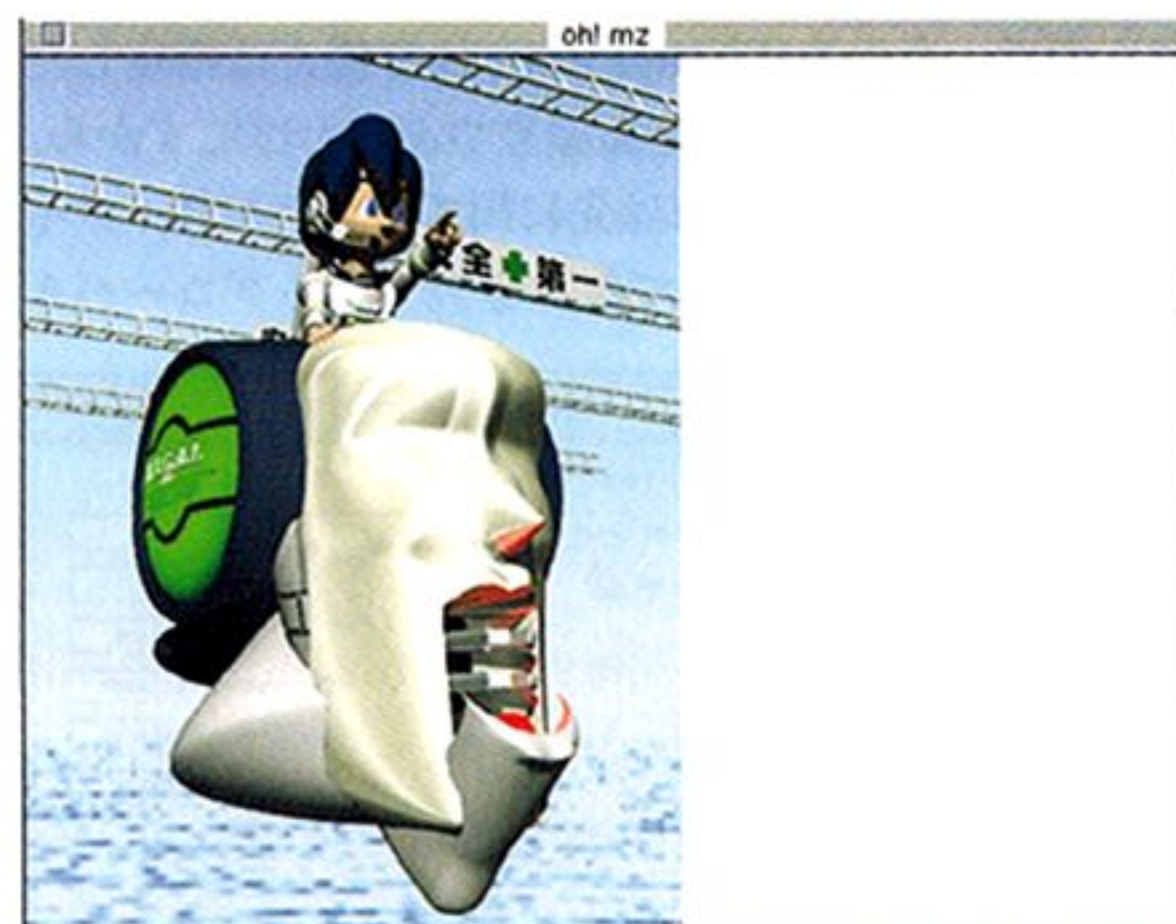



図16

ません。

GWorldの作成がうまくいくと，今度はpictIDをチェックし，－1ならそのまま帰ってしまいます。これは，なにも描画されていない真っ白なままの仮想画面がほしかったためです。

次に登場するのが，GetGWorld () 関数です。これはオフスクリーンGWorld版のGetPort () そのもので，現在描画対象になっているGWorldの情報を入手します。また，LockPixels () はHlock () と，SetGWorld () もSetPort () と対応しています。このあたり，前節のウィンドウへの描画とほとんど同じです。以下のようなかたちで覚えてしまいましょう。ほとんど定型句です。

```
GetGWorld ( &GWorld退避用変数 , &デバイス退避用変数 ) ;
LockPixels ( GetGWorldPixMap ( ＊描画したいGWorldへのポインタ )) ;
SetGWorld ( ＊描画したいGWorldへのポインタ , NIL ) ;

/＊ここにオフスクリーンGWorldへの描画処理などが入る＊/

UnlockPixels ( GetGWorldPixMap ( ＊描画したいGWorldへのポインタ )) ;
SetGWorld ( GWorld退避用変数 , デバイス退避用変数 ) ;
```

仮想画面を確保するというと，厄介な処理が並んでいるような気がしますが，実際はNewGWorld () 関数だけで確保できるもので，initOffScreen () 関数のほかの部分はそこに絵を描くための下準備と後始末に費やされています。その下準備も，見てみれば以前出てきたものの焼き直し程度です。

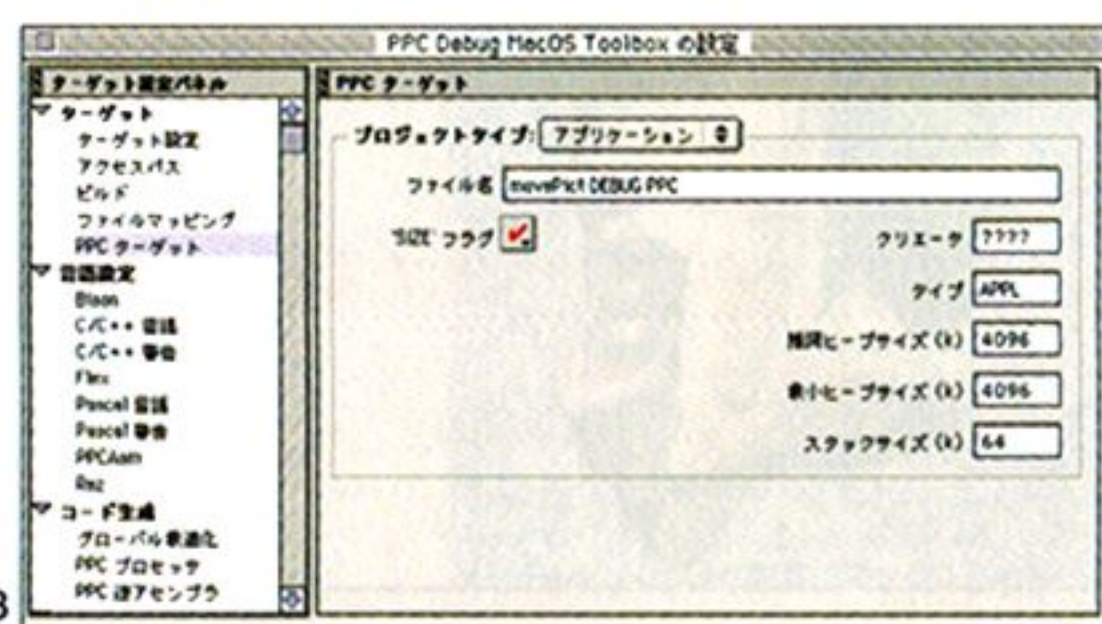
ではdrawObject () 関数に行きましょう。

さっそく新しい関数であるSetRect () が出てきました。これはRect型の変数に値を代入するための関数，offsetRect () は，Rect型で示されている長方形の領域を指定した数値だけ移動させる関数です。

次に現れたのがCopyBits () 関数。わかってしまえば簡単なのですが，引数が多く少し理解しにくいので図19を用意しました。関数表と一緒に見ればわかりやすいと思います。このCopyBits () 関数は，2つのウィンドウやGWorld間の長方形の領域を別のウィンドウやGWorldにコピーしてくれるだけでなく，長方形の大きさや形状が異なっていても，2つのGWorld (あるいはウィンドウ) 間に色深度の差があっても，きっちりと処理してくれる頼りになる関数です。

CopyBits () 関数は，このなかではキャラクターを仮想画面に描くために使われています。キャラクター用を表示させるためにわざわざオフスクリーンGWorldを用意して，そこから仮想画面へ描画するという方法を取っているのは，いちいちリソースから絵を読み直して仮想画面へ描き込むより手間が少なくて簡単だからです。また，調べたわけではありませんが，おそらくこのほうが数段速いでしょう。

最後はupdateScreen () 関数です。

やっていることはCopyBits () 関数を使って仮想画面から実画面へ画面を転送しているだけです。最後に，転送し終わった仮想画面の掃除をするために，EraseRect () 関数を使っています。これは，指定されたRect型の内部を，背景色で塗りつぶすものです。使っていなかったので，ここまで触れてきませんでしたが，QuickDrawは初期化された段階で，背景色を白に，描画色を黒に設定します。このプログラムでは，初期化したあと背景色も描画色も変更していませんので，そのままEraseRect () を使えば指定した領域を白く塗りつぶしてくれます。裏画面を全面塗りつぶすわけですから，指定する領域はmainWindowRectです。これは先に出てきたとおり，メインウィンドウの大きさがそのままRect型として保存されているグローバル変数です。

このあたりのことを図20にまとめましたので，ソースリストと突きあわせて調べてみてください。

前後を囲むGetGWorld (), SetGWorld () などについては，initOffScreen () 関数のときの説明のとおりです。

うはー，ずいぶんと詰め込みましたが，無事終了です。今回は仮想画面を使ったチラツキのない画面表示と，キャラクターの表示を一気に終わらせました。どちらもオフスクリーンGWorldを使うためにしかたなくこうなったのですが，一気に理解しようとせず，まず動かしていろいろ試してみてください。

さあ，これでついにキャラクターが動き出しました。movePict.cはいろいろと使える関数が詰まったソースです。自分なりに解析してぜひ自分のゲームに応用してみてください。

では，残るキーボードに行ってみましょう。準備はいいですか？

※15　本来なら，きちんとエラー処理をするか，どんな問題が起こったのかダイアログなどを表示すべきなのですが，今回の記事ではそのあたりも全部ばっさりと省略してあります。次の機会があれば，このあたりも説明したいなぁ……。

※16　フルカラー640×480ドットの仮想画面を1枚確保するごとに，約1Mバイト (＝1024Kバイト) 増やしていくのがひとつの目安です。もちろん，音などが入った場合は，そのことも考慮する必要があります。

※17　「情報を見る」で確認すると，ヒープサイズがターゲット設定パネルで指定したよりわずかに多い値になっています。ヒープサイズに実際のプログラムのサイズを足しあわせた値に近いようです。

マトリクス・キー・マトリクス

前節までで，今回の目標としているシューティングゲーム作成のための4つの目標のうち，3つ目まではクリアすることができました。最後の難関は，プレイヤーの押したキーを調べることです。この節のサンプルは，そういう意味では実用的なのかもしれません[※18]。

簡単そうで，本当に簡単で，実は意外な難関でもあるキーチェック。なにがどう簡単なのか，難しいのか，ぼそぼそと考えつつ最後の扉に手を伸ばすことにしましょう。

1）アプリケーションの実行まで

プロジェクト名はcheckKey.prj，リソース名はcheckKey.rsrcです。リソースは前回と同じものをコピーしてもかまいませんが，いらないリソースが入っていると気分が悪いので，使わないPICTリソースは削除してしまいましょう。

ソースそのものは，かなりの部分はコピー＆ペーストできます。initToolBox () とupdateScreen () はそのまま使っても大丈夫ですが，main関数は細かいところが変わっていますので注意してください。

また，仮想画面を使うために，ターゲット設定パネルを呼び出してアプリケーションの割り当てメモリを4096Kバイトに増やしておきましょう。多すぎるくらいでちょうどよいでしょう。気が向いたら，ファイル名を変えておきます。今回はkeyCheckerとしました。

dstBits　srcBits　srcRect　dstRect

src側とdst側のRectの大きさが異なれば，それにあわせて拡大縮小してくれる。また，色深度やカラーテーブルの異なるPixMap間でもキチンと処理してくれる。

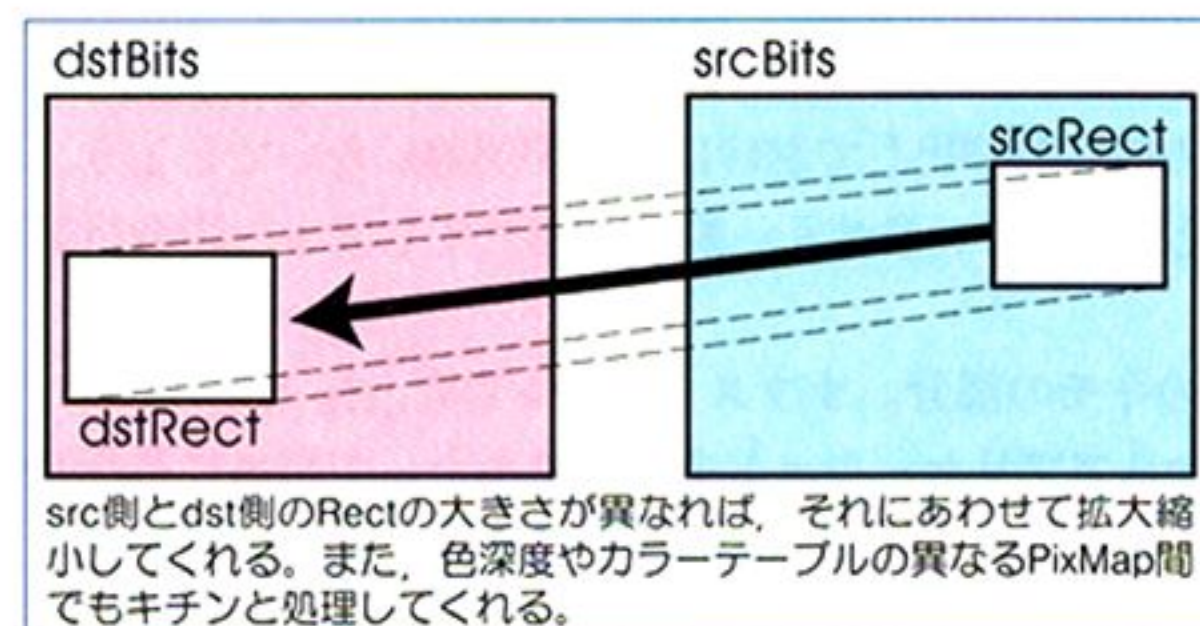
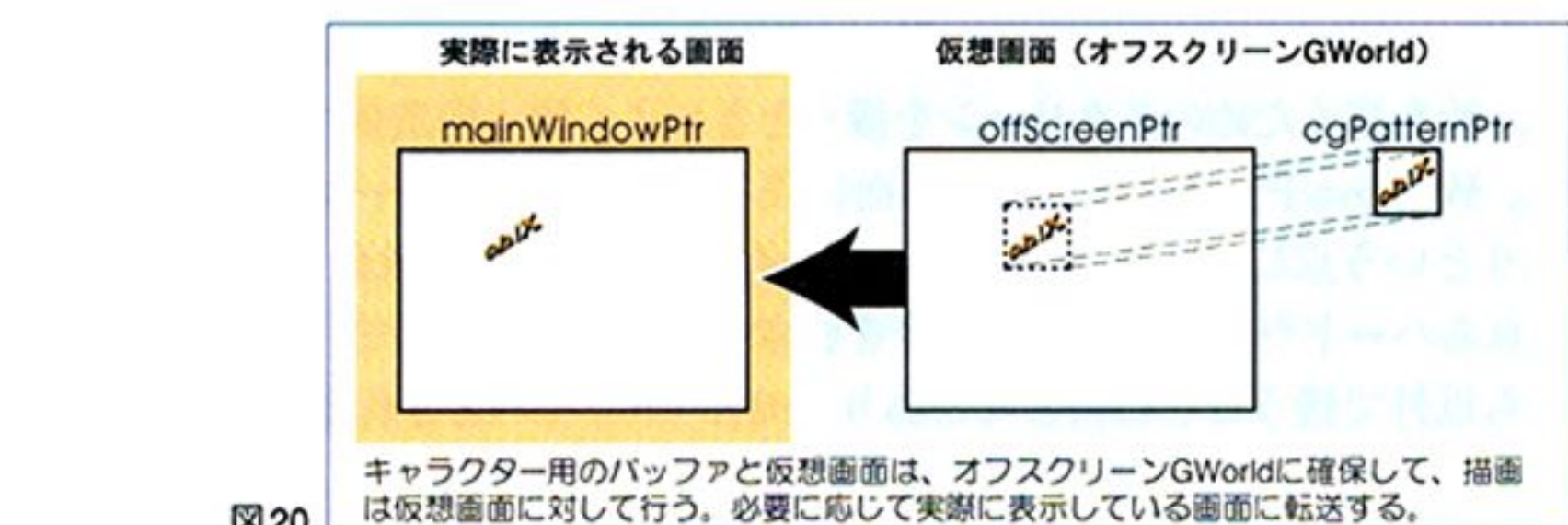
図19

実際に表示される画面　仮想画面（オフスクリーンGWorld）
mainWindowPtr　offScreenPtr　cgPatternPtr

キャラクター用のバッファと仮想画面は，オフスクリーンGWorldに確保して，描画は仮想画面に対して行う。必要に応じて実際に表示している画面に転送する。

図20

図17

図18

最後のサンプルプログラムです。慎重にいきましょう。

入力できましたか？　チェックは終わりましたか？　エラーは取れましたか？　では実行です。

2）出てきた数値はなんでしょう？

実行するとウィンドウが開き，0から15までの数字の横に0がずらっと並んでいます。もし，capslockが入っていれば，7番の横に2という数字が表示されているはずです。パワーキーは別にして，いろいろなキーを押し，どの番号の横にどんな数値が表示されたか確かめていきましょう。また複数のキーを押したときの値も見るのも忘れずに。

さて，このKeyChecker，いままでのサンプルとは違って，かなり実用的な意味も持っています。シューティングゲームやアクションゲームを作るにあたって，リアルタイムでのキーボードの情報はノドから手が出るほどほしいもののひとつ。複数キーをきちんと識別してくれるとなればなおさらです。

そこでリアルタイムでキーボード情報を取り込み，それを画面に表示するアプリケーションとして作成したものが，このKeyChecker。最終的に登場するSimpleShotのキー操作のための情報も，このKeyCheckerで得たものなのです。

3）checkKey.cの中身を見る

新しいのはcheckKey ()関数だけ，main ()も大きな変更はありません。initToolBox ()とupdateScreen ()はそのままです。

説明も短め，一気に行きましょう。

main ()で目新しいのは，FlushEvents ()関数です。

KeyCheckerはイベントに対応したプログラムではないので，作動しているあいだ，アプリケーションに渡されるはずのイベントがどんどん溜まっていきます。それまで溜まっていたイベントが全部Finderやほかのアプリケーションに吐き出されることになるために，どんな誤動作を引き起こすかわかりません[※19]。そこでイベントキューと呼ばれる，イベントを溜めておくバッファをクリアすることで，ほかのアプリケーションが誤動作を起こすことを防ぎます。

キー入力を実際に受け付け，状態を表示するのがcheckKey ()関数です。

まず目立つのがunsigned charの配列です。この配列のなかにキーボードからの情報が収められることになります。

Str255型は，Pascal型の文字列を示すものです。Cの文字列と違って，255文字までしか入りませんが，初期のMacOSがPascalで開発されていた影響が残っているのか，ToolBoxではいまだに多用されています。

関数の最初にきているGetGworld ()とSetGWorld ()に関しては以前の説明のとおりです。

次の行にあるGetKeys ()関数が今回のミソです。関数表を見るとわかるとおり，GetKeys ()は，呼び出された時点で押されているキーの状態をKeyMap型の変数に入れてくれるのですが，KeyMap型は結局16バイトの配列なので，このようなかたちで呼び出せば問題なく結果を受け取れます。

GetKeys ()から受け取った16バイトの配列の結果を表示するのが，その下のループです。

ループの中にある3つの関数はいずれも新顔です。順を追って説明しましょう。

NumToString ()関数は，1番目の引数として与えられた値をPascal型

リスト5　checkKey.c

```
#define REMOVE_EVENTS   0

#define kWindow_ID      200
#define kIN_FRONT       (WindowPtr)-1
#define NIL             0

/* 関数プロトタイプ */
void main(void); /* メイン関数 */
void initToolBox(void);/*ToolBox初期化*/
void checkKey(void);/* キーの状態をチェックして画面に描く */

void updateScreen(void);/* 裏画面を表画面に転送する。裏画面は塗りつぶして掃除する */

/* グローバル変数 */
WindowPtr mainWindowPtr;/*Window構造体を保存するためのポインタ*/
GWorldPtr offScreenPtr; /* 仮想画面用オフスクリーンGWorld*/

Rect mainWindowRect;/* ウィンドウの大きさ */

/* メイン関数 */
void main(void)
{
  initToolBox();/*ToolBox初期化*/

  /* リソースの情報をもとにウィンドウを作る */
  mainWindowPtr = GetNewCWindow( kWindow_ID, NIL , kIN_FRONT );
  if( mainWindowPtr == NIL )
  {
    ExitToShell();/* 失敗したら、なにもせずにFinderへ戻る */
            }

  /* ウィンドウの大きさを取り出しやすくするために、グローバル変数に入れておく */
  mainWindowRect = mainWindowPtr->portRect;

  /* オフスクリーンGWorldを作る */
  offScreenPtr = NIL;
  if( NewGWorld( &offScreenPtr , NIL , &mainWindowRect , NIL , NIL , NIL ) != noErr)
  {
  ExitToShell();    /* 確保できなかったら、終了する */
  }

  SysBeep(50);

  /* マウスボタンが押されるまで待つ */
  while (!Button())
  {
    checkKey();
    /* キー情報を表示 */
    updateScreen();
  };

  FlushEvents( everyEvent ,REMOVE_EVENTS );/* イベントキューのクリア */
  /* ほぼ定型句。定数定義と組でそのまま流用できる。*/
  /* 悪夢が見たい方は、この上のFlushEvents()をコメントアウトして、*/
  /* エディタを動かしたうえで、KeyCheckerを起動させてみましょう。*/

  return;/*        終わり */
}

/* 初期化 */
void initToolBox()
{
  /* 意地でもアプリケーションヒープを最大にする */
  MaxApplZone();

  /* 必要な各種マネージャの初期化。*/
  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(nil);
  InitCursor();
}

/* キーの状態を見る */
void checkKey(void)
{
  unsigned char status[16];/* キー情報入手用 */
  int i0;/* カウンタ */
  Str255 tempStr;/* パスカル型文字列 */

  //一時保存用
  GWorldPtr saveGWorldPtr;
  GDHandle saveGDHandle;

  GetGWorld( &saveGWorldPtr , &saveGDHandle );/* 元のGWorld保存 */
  SetGWorld( offScreenPtr , NIL );/* 確保したGWorldを描画ポートに設定する */

  GetKeys( (long *)status );/* キー情報の入手 */

  for( i0 = 0 ; i0 < 16 ; i0++ )
  {
    NumToString((long)i0, tempStr);// 数値を文字列に変換する
    MoveTo( 4 , i0*16+16 );//描画位置を決める
    DrawString( tempStr );

    NumToString((long)status[i0], tempStr);// 数値を文字列に変換する
    MoveTo( 32 , i0*16+16 );//描画位置を決める
    DrawString( tempStr );
  }

  SetGWorld( saveGWorldPtr , saveGDHandle );/* 描画ポート復旧 */

}

/* 裏画面を表画面に転送する。裏画面は塗りつぶして掃除する */
void updateScreen(void)
{
  //一時保存用
  GWorldPtr saveGWorldPtr;
  GDHandle saveGDHandle;

  //  メインスクリーンに絵を転送する
  CopyBits( &((GrafPtr)offScreenPtr)->portBits,/* 転送元GWorldまたはWindowPtr*/
    &((GrafPtr)mainWindowPtr)->portBits,/* 転送先GWorldまたはWindowPtr*/
    &mainWindowRect,/* 転送元Rect*/
    &mainWindowRect,/* 転送先Rect*/
    srcCopy,/* 転送方法(ベタ転送)*/
    NIL);/* マスク用Region、マスクしないのでNIL*/

  GetGWorld( &saveGWorldPtr , &saveGDHandle );/* 元のGWorld保存 */
  LockPixels( GetGWorldPixMap( offScreenPtr ));/* 描画のためのピクセルロック */
  SetGWorld( offScreenPtr , NIL );/* 確保したGWorldを描画ポートに設定する */

  EraseRect(&mainWindowRect);//ウィンドウ内を塗りつぶしておく。

  UnlockPixels( GetGWorldPixMap( offScreenPtr ));/* ピクセルロック解除 */
  SetGWorld( saveGWorldPtr , saveGDHandle );/* 描画ポート復旧 */
}
```

文字列に変換して2番目の引数に入れてくれます。ゲームであれば，スコアの表示など実にいろいろと使える関数です。

MoveTo()関数はQuickDrawの描画開始位置を設定します。

DrawString()関数は与えられたPascal型文字列をカレントポートに描き出します。

結局，このループでなにをしているかというと，GetKeys()関数から受け取った配列のなかの数値をNumToString()で文字に変換して，MoveTo()で位置決めをして，DrawString()関数で描き出しているだけなのです。

これで，ゲームに必要な基礎となるべきところはすべて習得しました。もう恐れるものはなにもありません。

次の節で登場するSimpleShotはいままで得た知識だけで完成できるMacOS用ゲームのひとつのサンプルです。

がんばって解析してみてください。

では，行きましょう。

※18　実際，これを作ったおかげでいろいろ楽ができました。
※19　前節まで作ってきたアプリケーションもイベントに対応してはいませんが，操作されることを前提にしたものではなく，動作をチェックしてマウスクリックで終了するだけのものだったので問題を起こすことは少ないだろうと棚上げにしてきた部分です。機会があれば，ちゃんとイベントに対応したアプリケーション開発の説明もやってみたいと考えています。

Hellow,Mr.SimpleShot!

さあ，SimpleShot.cを確認してみましょう。これが今回目標としていた，SimpleShotの全ソースです。ソースは長めですが，ほとんどの部分は実際にキャラを動かしたりするためのものです。

ここまでくればもう詳しい説明もいらないでしょう。さっそく入力です。

プロジェクト名はSimpleShot.prj，リソース名はSimpleShot.rsrcです。リソース内には必要なリソースがすべて収められています。ターゲット設定パネルを開き，クリエータが「7bBe」になっていることを確認してください。これはAppleに申請した，SimpleShotのための正式なクリエイタコードです。ヒープサイズは最小，推奨ともに4096になっているか確かめてみてください。

新しい関数が2つ3つ登場していますが，ソースリストのなかに十分コメントをとってありますので，別表の関数表と見比べて解析してみてください。特にCopyMask()関数は，背景つきのゲームには必須の関数になるでしょう。

実際の動作画面を図21に示します。

表5　マトリクス・キー・マトリクス

```
/*イベントキューからイベントを取り除く*/
void FlushEvents( short whicthMask , short stopMask );
引数：whitchMask 取り除くイベントの種類を決める。普通はeveryEvent（システム定義定
      数）を使ってすべてのイベントを取り除く
    ：stopMask 取り除くのを中止する条件となるイベント。普通は0（今回は
      REMOVE_EVENTSとして定義）を指定してすべてのイベントを取り除く

/*いま押されているキーの状態を得る*/
void GetKeys(KeyMap theKey);
引数：theKey いま押されているキーの状態が入る
返り値：なし

/*与えられた数値を文字列に変換する*/
void NumToString( long theNum , ConstStr255Param theString );
引数：theNum
文字列に変換したい数値：theString
変換した文字列を入れるためのPascal型文字変数：Str255型変数を渡せば問題ない
返り値：なし

/*描画開始位置の決定*/
void MoveTo( short x , short y );
引数：x
描画開始位置のX座標：y
描画開始位置のY座標
返り値：なし

/*Pascal文字列を描く*/
void DrawString(ConstStr255Param theString );
引数：theString
描画したい文字列（Pascal型文字列）
返り値：なし
```

1）SimpleShotの遊び方（1）

起動直後からゲーム開始です。カーソルキーで左右に移動，Commandキーで弾を発射します。敵も自機も弾に当たると爆発しますが，敵はランダムな位置に復活。自機は必ず画面中央に復活します。エスケープキーまたはQキーを押すか，マウスクリックで終了です。点数およびデモ，ゲームオーバー類は一切ありません。

2）SimpleShotの遊び方（2）

まず，画面の色深度をいろいろ変えて遊んでみてください。どの色深度でいちばん速度が稼げるか確かめてみるのが自作ゲームでスピードアップを図るための第一歩です。

次はソースの改造です。いろいろやって遊べるとは思いますが，いくつかヒントを。

まず，このソースリストのままでは，自機の移動はカーソルキーで操作していますが，これをテンキーに変更してみましょう。これは前節のKeyCheckerでキーの状態を調べて，どこをどう変更すればよいのか考えればよいでしょう。

命中判定を変えてみるのもひとつの手です。シビアにしたり，ルーズにしたりして遊びやすい命中判定を考えてみましょう。

ゲームスタートやゲームオーバーをつけたり，得点をつけることもできます。たくさんのキャラを同時に出すためにはどうすればよいのかも考えて，ぜひ試してみてください。

終わりに

「MacOSでゲームを作るのは難しくないんだ！」と勢いだけで書き始めて，一気にここまできたわけですが，見直してみると，やはり積み残したことが山のようにあります。イベントやさまざまなシステム定義定数の値や意味，構造体や変数型，効果音やBGMの処理やCarbon化，自作アプリケーションに自前のアイコンをつける作業についても詳しい説明をしていませんし……次の機会には，まずはイベントに対応させることを考えつつ，なにをしようか考えております。

しかしながら，シューティングに限らず，ゲームのもっとも肝心な部分，画面関係のことに関していえば，今回の内容で十分面白いものが作れるはずです。もし，それ以上に求めるものがあるとすれば，アイデアと努力で実

表6　Hellow,Mr.SimpleShot!

```
/*キーボードのFEPを切り替える*/
void KeyScript( short code );
引数：code 設定するスクリプトコードか切り替え方法を指定する。普通はsmRomanを指定してFEPを無
効化するのに使う（smRomanはシステム定義定数）
返り値：なし

/*マスターポインタブロックを追加する*/
void MoreMasters(void);
引数：なし。プログラムの最初のほうで呼んでおく必要がある。マスタポインタはプログラム実行中に必要
      に応じて追加されるが，この関数は前もってマスタポインタを追加しておくことでメモリの断片化を
      防ぐ働きがある
返り値：なし

/*システム内部時計を見る*/
void GetDateTime(unsigned long *sec);
引数：sec 1904年1月1日から現在までの秒数が入るが，そのままで使うことはまずない。SimpleShotの
      場合は，この値をシステムグローバル変数であるqd.randSeedに代入して，乱数の初期値を引っ掻
      き回している
返り値：なし

/*描画色を決める*/
void ForeColor( RGBColor theColor );
引数：QuickDrawの描画色を決める。SimpleShotで使われているwhiteColorやblackColorはシステム定
      義定数
返り値：なし

/*描画色を決める*/
void BackColor( RGBColor theColor );
引数：QuickDrawの背景色を決める。EraseRect()関数などで使われる
返り値：なし

/*マスクつきの画像の転送を行う*/
void CopyMask( const BitMap srcBis , const BitMap mskBits , const BitMap dstBitMap ,const Rect
*srcRect , const Rect *mskRect , cosnt Rect *dstRect );
引数：srcBits
転送元GWorldまたはWindowPtr：mskBits
マスク用GWorldまたはWindowPtr：dstBits
転送先GWorldまたはWindowPtr：srcRect
転送元Rect：mskRect
マスク用Rect：dstRect
転送先Rect
返り値：なし
```

現していくしかありません。

しかし趣味として純粋に楽しむためにプログラムを組んでいるわけですから，プロの方のようにあれこれ思い悩んだり，さまざまなガイドラインやルールに縛られる必要もありません。「暴走して当たり前」「ハングアップして当たり前」の気持ちでいろいろ試していきましょう。

また最初に書いたとおり，MacOS用のC言語の本には実践的なものは少なく，それらを探し出すことは至難の業ですが，逆に教科書的な良書は多く，大変に役立ちます。本記事でわからないことがあったら，そんな本で調べてみることをおすすめします。

次は……やっぱり横シューティングにします?!

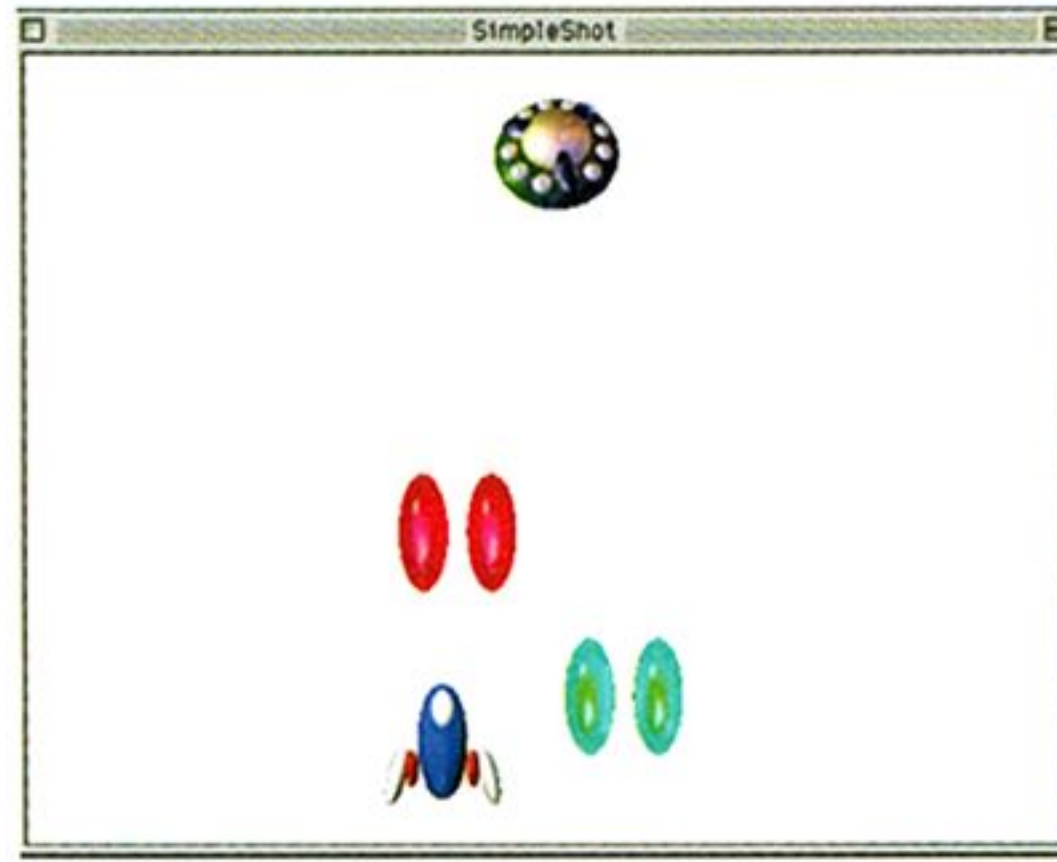



図21

リスト6 simpleshot.c

```
/*------------------------------------------------------------------------

  SimpleShot 00.1
  異常が発生したら、とにかく問答無用で終了するようになっているので、教材としてはあんまし頭が良く
ない。
  とりあえず、動くだけ。

------------------------------------------------------------------------*/

#include <Types.h>
#include <Memory.h>
#include <Quickdraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Menus.h>
#include <Windows.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <OSUtils.h>
#include <ToolUtils.h>
#include <Gestalt.h>

//------------------------------------------------------------ 定数の設定
/*必ず使う定数*/
#define REMOVE_EVENTS              0
#define NIL                        0L
#define IN_FRONT                   (WindowPtr)-1

/*キー入力関係*/
#define KEY_RIGHT                  0x01
#define KEY_LEFT                   0x02
#define KEY_FIRE                   0x04
#define KEY_QUIT                   0x08

/*キーに関する諸々……*/
#define RESULT_SHOT                0x01

/*moveObjAutoで、弾を撃つ確率（1/kProbShot）*/
#define kProbShot                  64

/*WindowのResourceID*/
#define kWindow_ID                 128

/*PictResource関係*/
#define FLYER_ID                   0
#define ENEMY_ID                   1
#define SHOT_ID                    2
#define BLAST_ID                   3
#define ESHOT_ID                   4

/*Objectの状態*/
#define OBJ_STANDBY                0
#define OBJ_LIVE                   1
#define OBJ_MOVE                   2
#define OBJ_EXPLODE                3

/*もろもろの定数*/
#define EXPLODE_TIME               50

/**/
#define GWColorDepth               16
//------------------------------------------------------------ グローバル変数の設定
/*  スタティック変数  */
Rect windRect;

/*画面表示用Window用領域*/
Rect  mainWindowRect;
WindowPtr  mainWindowPtr;

/*オフスクリーン用領域*/
Rect  offScreenRect;
GWorldPtr  offScreenPtr;

/*CGパターンバッファ*/
GWorldPtr  cgPatternPtr[5],cgMaskPtr[5];

//------------------------------------------------------------ 構造体の設定
//  typedefを使って定義しないのは、自分がどんな構造体を使って、何をしているのか見失わないため。

struct  objects{
    int LDflag;/*生死フラグ*/
    int nowX/*現在の座標*/,nowY;
    int targetX;/*目標座標とか……*/
    int targetY;
    int counter;/*カウンタ*/
    int U0;/*予備(^^;*/
    };

//------------------------------------------------------------ 関数プロトタイプ

/*標準手続き*/
void                       main(void);

/*utility関数群*/
void                       initToolBox(void);
int                        getKey(void);
void                       initOffScreen(GWorldPtr *,Rect *,int,int);
void                       drawPictFromRsrc( int );
void                       drawObject(int,int,int);
void                       updateScreen(void);

/*いろいろ初期化*/
void                       initWindow(void);
void                       initCharacter(void);

/*ゲーム本体とか…*/
void                       gameBody(void);
void                       initObjects(struct objects *);
int                        moveObjManual(struct objects *,int);
int                        moveObjAuto(struct objects *);
int                        objCollisionCheck(struct objects *,struct objects *);

/*異常終了時*/
void                       terminate(void);

/*デバッグ用*/
void                       flashInt(int);

//------------------------------------------------------------ main
//                         メイン。
//  これだけ
//  引数                   ：なし
//  戻り値                 ：なし
void main(void)
{
    initToolBox();/*ToolBoxの初期化*/
    initWindow();/*ゲーム用のウィンドウの初期化*/
    initCharacter();/*キャラクタの初期化*/

    KeyScript( smRoman );
    /*キースクリプトをシステムスクリプトに設定（FEP無効化）*/
                    /*ほぼ定型句。そのまま流用できる。*/

    FlushEvents( everyEvent ,REMOVE_EVENTS );/*誤動作を防ぐためのイベントキューのクリア*/
                    /*ほぼ定型句。定数定義と組でそのまま流用できる。
    */gameBody();

    FlushEvents( everyEvent ,REMOVE_EVENTS );/*誤動作を防ぐためのイベントキューのクリア*/
}

//------------------------------------------------------------ gameBody
//  ゲーム本体
//  とりあえず、格好だけでもついていればなんとか動くもんだ。
//  引数：なし
//  戻り値：なし
void gameBody()
{
    struct objects  flyer,enemy,fShot,eShot;

    int  keyState;/*キーの状態*/
    int  results;/*いろいろな返り値*/

    //一時保存用
    GWorldPtr  saveGWorldPtr;
    GDHandle  saveGDHandle;

    /*構造体の初期化*/
    initObjects( &flyer );
    initObjects( &enemy );
    initObjects( &fShot );
    initObjects( &eShot );

    /*キャラの初期化*/
    flyer.nowX = 224; flyer.nowY = 304; flyer.LDflag = OBJ_LIVE;
    enemy.nowX = 224; enemy.nowY = 16; enemy.LDflag = OBJ_LIVE;

    /*ループ始まり*/
    while(true)
{
    keyState = getKey();/*キーの状態を得る*/
if((keyState & KEY_QUIT)||(Button())){ break; }/*マウスのボタンが押されるか、終了キーが押されていたら
終わり*/

results = moveObjManual( &flyer , keyState );/*flyer移動（位置チェック込み）*/

if(( results & RESULT_SHOT )&&( fShot.LDflag != OBJ_LIVE ))/*flyer弾発射（発射条件に注意！）*/
{
    fShot.LDflag = OBJ_LIVE ;/*発射フラグセット*/
    fShot.nowX =flyer.nowX ;/*位置はflyerからもらう*/
    fShot.nowY = flyer.nowY ;}
}

/*弾発射してる？*/
if( fShot.LDflag != OBJ_STANDBY )
{
```

リスト6 SimpleShot.c

```
/*画面からはみ出たら弾を消す*/
    if( fShot.nowY >= 8){ fShot.nowY -= 8;}else{ fShot.LDflag = OBJ_STANDBY;}
    /*命中判定（位置及びenemyFlagで爆発していないかのチェック）*/
    if((objCollisionCheck( &fShot , &enemy) == -1)&&(enemy.LDflag != OBJ_EXPLODE ))
    {
        enemy.LDflag =3;/*爆発中フラグセット*/
        enemy.counter = EXPLODE_TIME;/*爆発している時間（増やすと伸びる）*/
        fShot.LDflag = OBJ_STANDBY;/*命中したショットは消える*/
    }
}

/*enemy移動（弾も撃つか考える）*/
results = moveObjAuto( &enemy );

/*enemy弾発射（発射条件に注意！）*/
if(( results & RESULT_SHOT )&&( eShot.LDflag != 1 )&&( enemy.LDflag != OBJ_EXPLODE ))
{
    eShot.LDflag = OBJ_LIVE ;/*発射フラグセット*/
    eShot.nowX =enemy.nowX ;/*位置はenemyからもらう*/
    eShot.nowY = enemy.nowY ;
}
/*このあたりは、上のflyerとenemyでやっている事はほとんど同じ。あえてサブルーチン化はしなかった。*/
/*いろいろ試したら、このサブルーチン化してflyerとenemyの処理を共通にしてみよう*/

/*弾発射してる？*/
if( eShot.LDflag != 0 )
{
    /*画面からはみ出たら弾を消す*/
    if( eShot.nowY < 384 ){ eShot.nowY += 8;}else{ eShot.LDflag = OBJ_STANDBY;}
    /*命中判定（位置及びenemyFlagで爆発していないかのチェック）*/
    if((objCollisionCheck( &eShot , &flyer) == -1)&&(flyer.LDflag != OBJ_EXPLODE ))
    {
        flyer.LDflag = OBJ_EXPLODE;/*爆発中フラグセット*/
        flyer.counter = EXPLODE_TIME;/*爆発している時間（増やすと伸びる）*/
        eShot.LDflag = OBJ_STANDBY;/*命中したショットは消える*/
    }
}

/*キャラを描く*/
if(fShot.LDflag != 0){drawObject( fShot.nowX, fShot.nowY , SHOT_ID);} /*弾を撃っていたら、弾を描く*/
if(eShot.LDflag != 0){drawObject( eShot.nowX, eShot.nowY , ESHOT_ID);} /*敵も弾を撃っていたら、描く*/

/*enemyは爆発してますか？*/
if(enemy.LDflag == OBJ_EXPLODE )
{
    drawObject( enemy.nowX+Random()%5, enemy.nowY+Random()%5,BLAST_ID);/*爆発してたら、
    爆炎を出す。*/
}
else
{
    drawObject( enemy.nowX , enemy.nowY , ENEMY_ID);/*enemyを描く*/
}

/*flyerは爆発してますか？*/
if(flyer.LDflag == OBJ_EXPLODE )
{
    drawObject( flyer.nowX+Random()%4, flyer.nowY+Random()%4,BLAST_ID);/*爆発してたら、爆炎を出す。*/
}
else
{
    drawObject( flyer.nowX , flyer.nowY , FLYER_ID);/*flyerを描く*/
}

/*デバッグ用*/
/*
GetGWorld( &saveGWorldPtr , &saveGDHandle );//元のGWorld保存
SetGWorld( offScreenPtr , NIL );//確保したGWorldを描画ポートに設定する

flashInt( enemy.LDflag );

SetGWorld( saveGWorldPtr , saveGDHandle );//描画ポート復旧
*/
/*画面アップデート*/
updateScreen();
}
}

//---------------------------------------------------------------- moveObjManual
//  オブジェクトをキー操作で動かす
//  引数：動かしたいobjects構造体へのポインタ
//        ：getKeyで入手したキーの状態
//  戻り値：弾を撃ったか？
int moveObjManual(struct objects *theObjects , int theKey)
{
    int   result;

    result = 0;/*戻り値を初期化*/

    switch(theObjects->LDflag)
    {
        case OBJ_LIVE:/*生きて動いてます*/
            /*flyer移動（位置チェック込み）*/
            if((theKey & KEY_LEFT)&&( theObjects->nowX >= 8)){ theObjects->nowX -= 2;}
            if((theKey & KEY_RIGHT)&&( theObjects->nowX <= 440)){ theObjects->nowX += 2;}

            /*flyer弾発射*/
            if(theKey & KEY_FIRE){ result != RESULT_SHOT; }
        break;

        case OBJ_EXPLODE:/*爆発してます（この間、身動きは取れない）*/
            if(theObjects->counter > 0){ (theObjects->counter)--; }/*爆発カウンタ減らす*/else
            {
            theObjects->nowX = 224; /*位置再初期化*/
            theObjects->LDflag = OBJ_LIVE;/*生き返る*/
            }

        default:
        break;
    }
        return(result);
}

//---------------------------------------------------------------- moveObjAuto
//  オブジェクトを勝手に動かす
//  引数：動かしたいobjects構造体へのポインタ
//  戻り値：弾を撃ったか？（撃つなら-1、撃たないなら0）
int moveObjAuto(struct objects *theObjects)
{
    int   result;

    result = 0;

    /*自動的に移動…*/
    switch( theObjects->LDflag )
    {
        case OBJ_LIVE:/*待機*/
            theObjects->targetX = Random()% 216 + 226; /*移動の目的位置設定*/
            theObjects->LDflag = OBJ_MOVE;
            /*移動中にする*/
        break;

        case OBJ_MOVE:/*移動中（目標座標に向けてずるずる動く）*/
            if( theObjects->nowX < theObjects->targetX ){ (theObjects->nowX)++; }
            else
            if( theObjects->nowX > theObjects->targetX ){ (theObjects->nowX)--; }
            else
            { theObjects->LDflag = OBJ_LIVE;}/*目標座標に着いたら、待機状態にする*/
        break;

        case OBJ_EXPLODE:/*爆発*/
            if(theObjects->counter > 0){ (theObjects->counter)--; }/*爆発カウンタ減らす*/
            else
            {
            theObjects->nowX = Random()% 216 + 226; /*移動の目的位置設定*/
            theObjects->LDflag = OBJ_LIVE;
            /*待機状態にする*/
            }
        break;

        default:
        break;
        }

    /*弾撃つか？（ここでは、撃てる状態かどうかはチェックしていない）*/
    if(Random() % kProbShot == 0){ result != RESULT_SHOT;}

    return(result);
}

//---------------------------------------------------------------- objCollisionCheck
//  objects構造体同士の衝突チェック
//  引数：衝突判定をしたい2つのobjects構造体へのポインタ
//  戻り値：衝突してたら-1、そうでなけりゃ0
int objCollisionCheck(struct objects *objectA,struct objects *objectB)
{
    /*結果の初期化*/
    int   result;

    if((objectA->nowX-32 < objectB->nowX )&&(objectB->nowX < objectA->nowX +32 )&&
      (objectA->nowY-32 < objectB->nowY )&&(objectB->nowY < objectA->nowY + 32 )){ result = -1;
      }else{ result = 0; }

    return(result);
}

//---------------------------------------------------------------- initObjects
//  objects構造体の初期化
//  引数：初期化したいobjects構造体へのポインタ
//  戻り値：なし
void initObjects(struct objects *theObjects)
{
    theObjects->LDflag = 0;
    theObjects->nowX = 0;
    theObjects->nowY = 0;
    theObjects->targetX = 0;
    theObjects->targetY = 0;
    theObjects->counter = 0;
    theObjects->U0 = 0;
}

//---------------------------------------------------------------- initToolBox
//  ツールボックスの初期化
//  引数：なし
//  戻り値：なし
void initToolBox()
{
    /*意地でもアプリケーションヒープを最大にする*/
    MaxApplZone();
    /*マスタ・ポインタを山ほど増やす*/
    MoreMasters();MoreMasters();MoreMasters();MoreMasters();MoreMasters();MoreMasters();More
    Masters();MoreMasters();
    MoreMasters();MoreMasters();MoreMasters();MoreMasters();MoreMasters();MoreMasters();More
    Masters();MoreMasters();

    /*必要な各種マネージャの初期化。*/
    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NIL);
    InitCursor();

    /*ランダマイズ（乱数をさらに引っ掻き回す）*/
    GetDateTime((unsigned long*) &qd.randSeed);

}

//---------------------------------------------------------------- initWindow
//  ゲーム用Windowを初期化する。オフスクリーン画面も確保する。
//  引数：なし
//  戻り値：なし
void initWindow(void)
{
    WindowPtr   oldWindowPtr;

    //ゲーム用ウィンドウのポインタを空にしておく。
    mainWindowPtr = NIL;

    //ゲーム用ウィンドウの生成。駄目だったら終わる
    mainWindowPtr = GetNewCWindow( kWindow_ID, NIL , IN_FRONT );
    if( NIL == mainWindowPtr ){ terminate();}
```

```
	ShowWindow( mainWindowPtr );/* いま作ったウィンドウを表示する */

	GetPort(&oldWindowPtr);/* 現在の描画用ウィンドウのポインタを保存する */
	SetPort(mainWindowPtr);/* 描画用ウィンドウをいま生成したウィンドウに設定する。 */

	ForeColor(blackColor);/* ウィンドウ内を真っ白に塗りつぶしておくために…… */
	BackColor(whiteColor);

	mainWindowRect = mainWindowPtr->portRect;/*Rect領域を取り出すためのTip（と言うほどのモンじゃない） */
	offScreenRect = mainWindowRect;

	EraseRect( &mainWindowRect );/*塗りつぶして初期化*/

	SetPort(oldWindowPtr);/* 描画用ウィンドウを元に戻す。 */

	/*オフスクリーンを生成する*/
	initOffScreen( &offScreenPtr , &offScreenRect , GWColorDepth , -1);

}

//---------------------------------------------------------------- initCharacter
//キャラクターの初期化
//	引数：無し
//	戻り値：無し
void initCharacter(void)
{
	Rect	tempRect;

	SetRect(&tempRect,0,0,64,64);/* キャラクターのサイズのRect*/

	/*キャラクターを取り込む*/
	initOffScreen(&cgPatternPtr[FLYER_ID],&tempRect,GWColorDepth, 200 + FLYER_ID);
	initOffScreen(&cgPatternPtr[ENEMY_ID],&tempRect,GWColorDepth, 200 + ENEMY_ID);
	initOffScreen(&cgPatternPtr[SHOT_ID],&tempRect,GWColorDepth, 200 + SHOT_ID);
	initOffScreen(&cgPatternPtr[BLAST_ID],&tempRect,GWColorDepth, 200 + BLAST_ID);
	initOffScreen(&cgPatternPtr[ESHOT_ID],&tempRect,GWColorDepth, 200 + ESHOT_ID);

	/*マスクを取り込む*/
	initOffScreen(&cgMaskPtr[FLYER_ID],&tempRect,1, 300 + FLYER_ID);
	initOffScreen(&cgMaskPtr[ENEMY_ID],&tempRect,1, 300 + ENEMY_ID);
	initOffScreen(&cgMaskPtr[SHOT_ID],&tempRect,1, 300 + SHOT_ID);
	initOffScreen(&cgMaskPtr[BLAST_ID],&tempRect,1, 300 + BLAST_ID);
	initOffScreen(&cgMaskPtr[ESHOT_ID],&tempRect,1, 300 + ESHOT_ID);
}

//---------------------------------------------------------------- initOffScreen
//	オフスクリーンのPixMapを作る。確保できなかった場合は、そのまま終わる
//引数：*theGWorldPtr 新たに作るGWorld構造体へのポインタ
//		：*tempRect 新たに作るGWorld構造体の領域（0,0から始まる必要はない）
//		：theDepth 新しく作るGWorldの色深度（モニタの色深度は気にしない……チェックしていないので
//		  不正な値を渡すと死ぬ）
//		：ImageID ロードすべきイメージのResourceID。もし-1なら、ロードせず白で塗りつぶす
//返り値：なし
void initOffScreen(GWorldPtr *theGWorldPtr,Rect *theRect,int theDepth,int ImageID)
{
	//一時保存用
	GWorldPtr saveGWorldPtr;
	GDHandle saveGDHandle;
	Rect tempRect;

	tempRect = *theRect;

	//Rectの左肩を 0,0 にあわせる
	OffsetRect( &tempRect,-tempRect.left, -tempRect.top);

	/*元のGWorld保存（作ったGWorldに画像をロードするため）*/
	GetGWorld( &saveGWorldPtr , &saveGDHandle );

	//	ＢＧ用オフスクリーン（NewGWorld）を確保できなかった場合は、異常終了
	*theGWorldPtr = NIL;
	if( NewGWorld( theGWorldPtr , theDepth , &tempRect , NULL , NULL , NULL ) != noErr ){
	terminate(); }

	LockPixels( GetGWorldPixMap( *theGWorldPtr ));/*描画のためのピクセルロック*/
	SetGWorld( *theGWorldPtr , NIL );/*確保したGWorldを描画ポートに設定する*/

	ForeColor(blackColor);/*描画色を黒に*/
	BackColor(whiteColor);/*背景色を白に*/

	if( ImageID != -1)
	{
		drawPictFromRsrc( ImageID );/*-1でなければ描画*/
	}
	else
	{
		EraseRect( &tempRect);/*tempRectの範囲（＝新しく確保したGWorld）を消去*/
	}

	UnlockPixels( GetGWorldPixMap( *theGWorldPtr ));/*ピクセルロック解除*/
	SetGWorld( saveGWorldPtr , saveGDHandle );/*描画ポート復旧*/
}

//---------------------------------------------------------------- drawPictFromRsrc
//	PICTリソースから絵を取り込んで、カレントポートの(0,0)表示する。
//	スケーリングはしない。
//	引数：表示させるPICTリソースのID
//	戻り値：無し
void drawPictFromRsrc(int resID)
{
	Rect tempRect;
	PicHandle tempPictHandle;

	tempPictHandle = GetPicture(resID);
	/* リソースフォークからPICTを読み込む。*/

	if (tempPictHandle == NIL){ terminate(); }/* 取り込めなかったら異常終了*/

	HLock( (Handle)tempPictHandle);
/*	リソース操作中にちょっかいを出されるのを防ぐ*/

	tempRect = (*tempPictHandle)->picFrame;/*	PICTの大きさを調べておく*/
	OffsetRect(&tempRect, -tempRect.left, -tempRect.top);/*	描画領域の左肩を 0,0 にあわせる（よく使う表現）*/

	DrawPicture( tempPictHandle, &tempRect);/*	カレントポート（それがOFFスクリーンでも構わない）にPICTを書き込む*/

	HUnlock( (Handle)tempPictHandle);/*	ロック解除*/

	ReleaseResource( (Handle)tempPictHandle);/* メモリを占有している絵（PICT）を破棄する。*/
}

//---------------------------------------------------------------- drawObject
//オフスクリーンにキャラクターを描く
//	引数：x0,y0 キャラクターの座標
//		：objType　キャラクターのタイプ
//	戻り値：無し
void drawObject(int x0,int y0,int objType)
{
	Rect	srcRect,dstRect;/*描画範囲を保持する変数。*/

	SetRect(&srcRect,0,0,64,64);/*領域を決める。*/
	dstRect = srcRect;

	OffsetRect( &dstRect , x0 , y0 );/*実際に書き込む位置にRectをオフセットする。*/

	CopyMask(&((GrafPtr)cgPatternPtr[objType])->portBits,/*転送元GWorldまたはWindowPtr*/
		&((GrafPtr)cgMaskPtr[objType])->portBits,/*マスク用GWorld（WindowPtrって事はないだろう）*/
		&((GrafPtr)offScreenPtr)->portBits, /*転送先GWorldまたはWindowPtr*/
		&srcRect,
		/*転送元Rect*/
		&srcRect,
		/*マスク用Rect*/
		&dstRect);
		/*転送先Rect*/
}

//---------------------------------------------------------------- updateScreen
//	裏画面から表画面に転送してアップデートする。
//	その際、裏画面は塗りつぶされる。
//	引数：なし
//	戻り値：なし
void updateScreen(void)
{
	//一時保存用
	GWorldPtr	saveGWorldPtr;
	GDHandle	saveGDHandle;

	//	メインスクリーンに絵を転送する
	CopyBits(&((GrafPtr)offScreenPtr)->portBits,/*転送元GWorldまたはWindowPtr*/
		&((GrafPtr)mainWindowPtr)->portBits,/*転送先GWorldまたはWindowPtr*/
		&offScreenRect,
		/*転送元Rect*/
		 &offScreenRect,
		/*転送先Rect*/
		srcCopy,
		/*転送方法(この場合はベタ転送)*/
		NIL);
		/*マスク用Region、今回はマスクしないのでNIL*/

	GetGWorld( &saveGWorldPtr , &saveGDHandle );/*元のGWorld保存*/
	SetGWorld( offScreenPtr , NIL );/*確保したGWorldを描画ポートに設定する*/

	EraseRect(&offScreenRect);
	//ウィンドウ内を塗りつぶしておく。

	SetGWorld( saveGWorldPtr , saveGDHandle );/*描画ポート復旧*/
}

//---------------------------------------------------------------- getKey
//キー入力情報を得るルーチン。
//	引数：無し
//	戻り値：押されているキーの情報をintに入れて返す

int getKey(void)
{
	unsigned char	status[16];
	long	key_status;

	key_status = 0;/*キーステータスの初期化*/
	GetKeys( (long *)status );/*キー情報の入手*/

	//キーのチェック//
	if ( status[15]  & 0x10 )	key_status |= KEY_RIGHT;/*rightキー（カーソルキー）*/
	if ( status[15]  & 0x08 )	key_status |= KEY_LEFT;/*leftキー（カーソルキー）*/

	if ( status[06]  & 0x80 )	key_status |= KEY_FIRE;/*cmdキー*/

	if ( status[06]  & 0x20 )	key_status |= KEY_QUIT;/*escキー*/
	if ( status[01]  & 0x10 )	key_status |= KEY_QUIT;/*"q"キー*/

	return( key_status );
}

//---------------------------------------------------------------- terminate
//	異常終了
//	引数：なし
//	戻り値：なし（帰らない…）
void	terminate(void)
{
	SysBeep(50);
	ExitToShell();
}

//---------------------------------------------------------------- flashInt
//	画面の左上に数値を書き込むルーチン。デバッグ用。
void flashInt (int theValue)
{
	Str255	tempStr;
	Rect	tempRect;

	NumToString((long)theValue, tempStr);// 数値を文字列に変換する
	MoveTo(32, 32);
	// QDペンの移動
	SetRect(&tempRect, 16, 16, 240, 48);// 領域を設定
	EraseRect(&tempRect);
	// 領域内を白くくりぬく
	DrawString(tempStr);
	// デバッグ用文字列を書き込む
}
```

コンポーネントを使ってHTMLエディタを作る

中野修一 Nakano Shuichi

VisualBasicではさまざまなコンポーネントを活用することができます。API呼び出しよりも簡単で，それに匹敵するくらい強力な処理を手軽に手に入れることができるのです。ただ，仕様の公開されてないコンポーネントは扱いづらいもの。ここでは導入の実際を手順を追って見てみましょう。

Webのお仕事

なんでもWebのお仕事だそうで，Oh!Xのほうもずいぶんわりを食ってしまったらしい。

ちなみに，Webサイトを作っているからといって，Webページとかに詳しい人ばかりが集まっていると思うのは間違いである。インターネット産業の中枢ソフトバンクといえど，もともと雑誌の編集者なのでざっと見て，ちゃんとタグを使えるのは6人中2人。これでなんとかなってるんだから文句をいう筋合いではないが，Webページの更新など，それなりに苦労してる人もいるみたいだ。

「そういうのは外注先のWebデザイン会社に任せればいいではないか」と考えるのはそれなりに自然な意見である。が，総合的に判断してなるべく任せないほうがよいという結論に達したようだ。少なくとも身内でやっていれば「背景画像に文章を書き込まないでください！」といった想像を絶した事態は起こらないし，やたらと画像化された文字の間違い修正のためにグラフィックエディタでドット修正するようなことはしなくても済む。新規デザインやデータコンバートを除いた日々の更新は内部作業になっているようだ。

Webページを作ることというかHTMLを書くこと自体は別に難しいものではないと思うのだが(小学生だってやってる)，慣れはある程度必要であり，HTMLへの多少の理解は不可欠だ。そういった部分はしかたないとしても，定型作業の部分は自動化できる可能性が多分にある。多少の省力化でもできればずいぶん効率も違ってくるはずなのだが。

そんなこんなで今回はHTMLを加工するためのツールを作ってみよう。

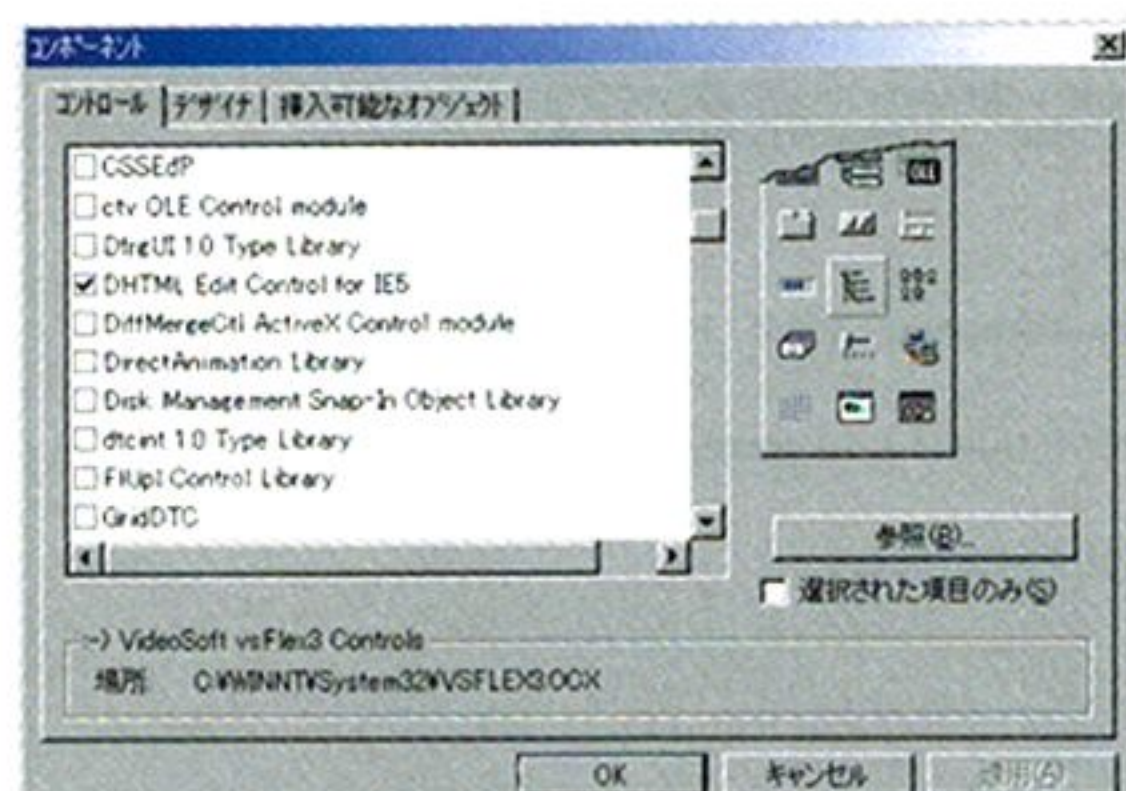



図1　コンポーネントから「DHTML Edit Control for IE5」を選択する

図2　新しく加わったコンポーネントをダブルクリックで呼び出そう

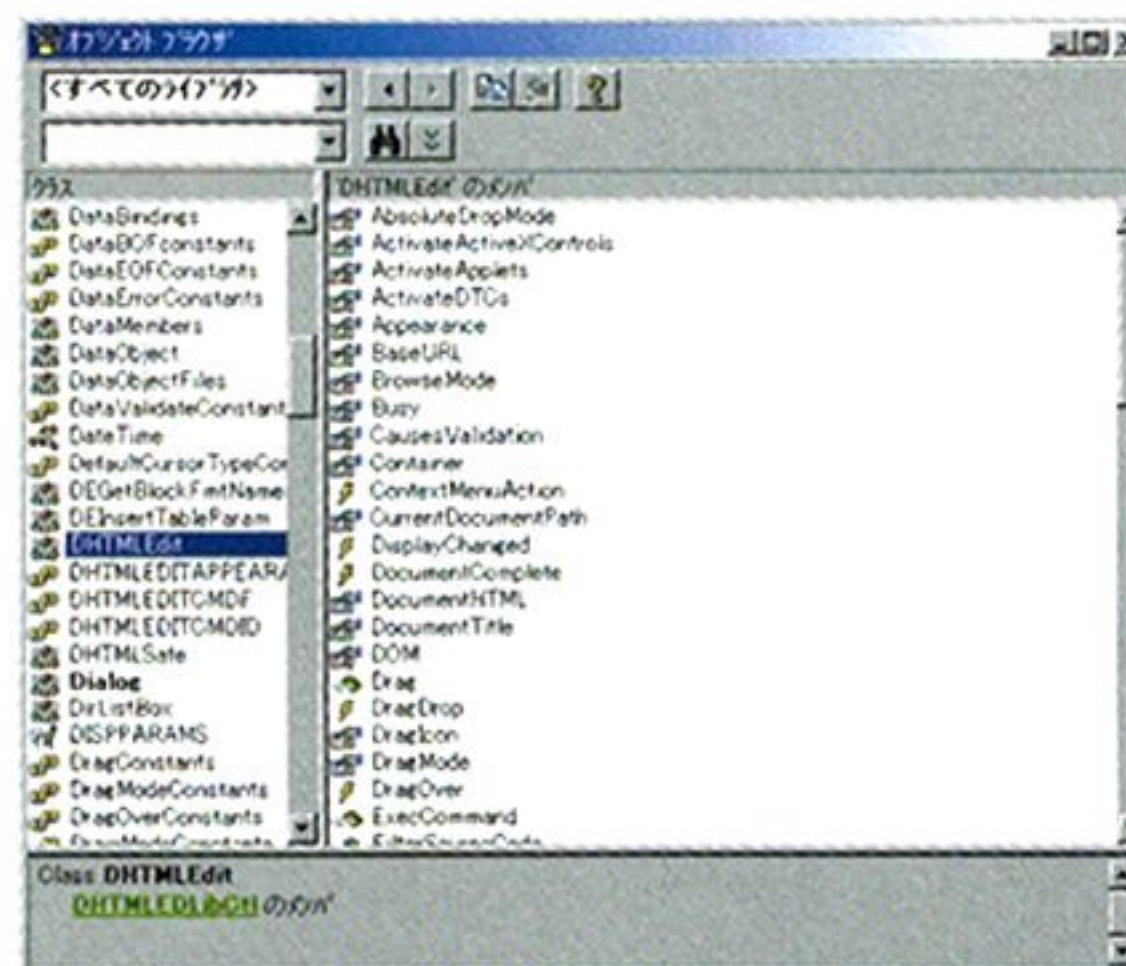



図3　オブジェクトブラウザで調べてみるとだいたいの使い方がわかる

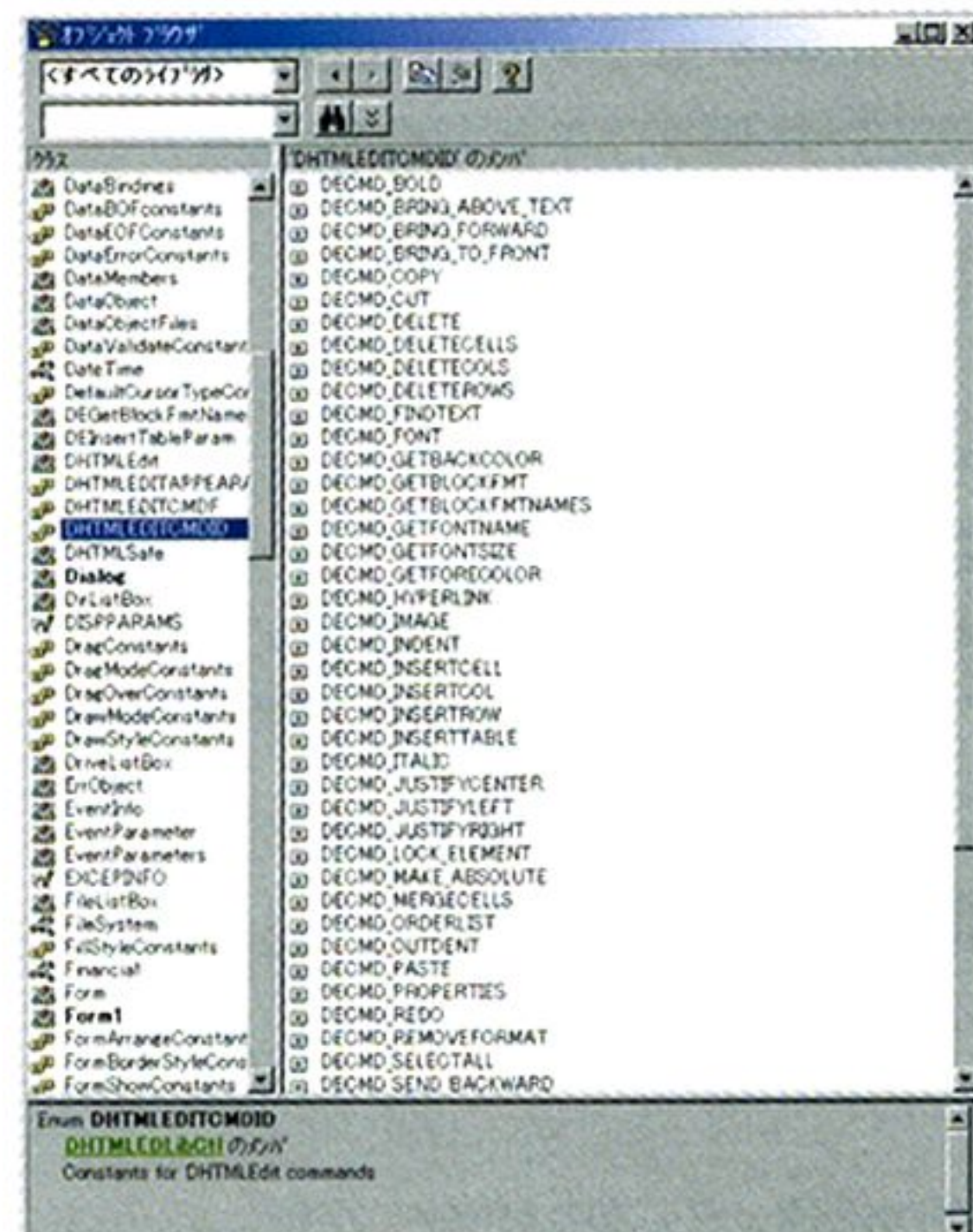



図4　これがExecCommandで使用できるコマンドの一覧だ

図5　こんな風に動く。上でGUI操作,下でソースレベルの処理を行う

「んなもん，テキストエディタと同じじゃん」という意見もあろうが，多少アプローチは変えてみたい。

ざっと仕様を考えてみる。テキストベースの処理ができないとまず話にならない。が，使うと想定される人のことを考えるとビジュアルである必要もある。VwalkerでWebデザインに使われている夢紡ぎ系のソフトは2KBのソーステキストを10KBのHTMLファイルに変換するという恐るべき能力を持つ。絡みあったテーブルタグ攻撃の前には，HTMLソースだけだと構造が追えない人も続出してしまう。日常の作業のほとんどは新規ファイルのアップロードとそれにともなうトップページの一部決まった位置の書き換え，書き足しと思われる。この部分がそれなりに問題になっているようだ。テーブルの1ブロックの差し替えとか，組み替えなどがわかりにくいのだ。こういったものはビジュアルというか，GUIでひょいと操作できないといけない。

ビジュアルにといっても，VBでタグを解釈して表示していては使いものにはなるまい。ということで，今回はDHTL Edit Control for IE5というコンポーネントを利用する。これはマイクロソフトがIE5のDHTML用に制作したコンポーネントで，IE5と同じモジュールを使い回しているだろうから，表示の面ではIE5と同等のHTML解釈が期待できる……などという甘いことは例によってないのだが，そこそこの表示は可能だ。IE5用のコンポーネントだからIE5のインストールで入るはずだ。

FrontPageなり，素直にそういうのを使えばいいじゃないかという意見もあるだろうが，それはそれ，彼らを甘く見てはいけない。ソースをなかなかグチャグチャにしてくれるツールが少なくないうえ，FrontPageだと最善を尽くしても1行につき2バイトずつ増えていくみたいだ(改行コードの変換ミスか？　編集するごとに半角スペースが溜まっていく)。

なんらかのアプリケーションをインストールしていくと，VBで使用できる高機能なコンポーネントも増えてくる。適度に手助けさせつつ，決して余計なマネはさせないようにするのが彼らとの賢いつきあい方である（いや，表示確認のためだけならこういうのを使わずいちいちブラウザを開いてもよかったのだが，なんとなく面白そうなので使ってしまった)。

よくわからないコンポーネントを使う

まず，コンポーネントを使用可能な状態にする必要がある。[プロジェクト]-[コンポーネント]を開いて，DHTL Edit Control for IE5のところにチェックマークをつけて[OK]ボタンを押す。ツールバーに現れたDHTML Editアイコンをダブルクリックすればフォーム内に取り込める。とはいえ，得体の知れないコンポーネントなので使いものになるのかどうかを見極めるのが先決だ。

オブジェクトブラウザを起動して，DHTMLEditオブジェクトの概要を見てみよう。ここでどんなプロパティがあるか，関数(メソッド)が用意されているのか，どういうイベントに対応しているのかがわかる。具体的にどんな機能があるのかはわからないが，あとは名前で類推していく。

表1をざっと見るとExecCommandなどという関数が見える。どんなコマンドが使えるのかというのは定数表を見ればわかる。この場合はDHTML EDITCMDIDだ。今回はあまり必要ではないので使用していないが，このコンポーネントからHTMLの簡易装飾などができるエディタを作る際には有用かもしれない。

とりあえず，重要そうなのは，

DocumentHTML

リスト

```
Option Explicit
Dim cr As String
Dim i As Integer, j As Integer
Dim url As String
Dim lof As Integer
Dim undo(10) As String
Dim udp As Integer
Dim baseurl As String
Public ss1 As String, ss2 As String

Dim sp1 As Long, sp0 As Long
Dim tp1 As Long, tp0 As Long
Dim f1 As Integer
Dim f2 As Integer
Dim temp1 As String
Dim temp3 As String
Dim temp2 As String
Dim hs As String
Dim ls As Integer
Dim lr As Integer
Dim rs As String

Private Sub Command1_Click()
udp = udp + 1
If udp > 10 Then udp = 1
undo(udp) = Text1.Text

Text1.Text = DHTMLEdit1.DocumentHTML
End Sub

Private Sub Command2_Click()
udp = udp + 1
If udp > 10 Then udp = 1
undo(udp) = DHTMLEdit1.DocumentHTML
DHTMLEdit1.DocumentHTML = Text1.Text
DHTMLEdit1.baseurl = baseurl
End Sub

Private Sub Command3_Click()
Text1.Text = undo(udp)
DHTMLEdit1.DocumentHTML = Text1.Text
DHTMLEdit1.baseurl = baseurl
udp = udp - 1
If udp < 1 Then udp = 10
End Sub

Private Sub Command6_Click()
Dialog.Visible = True
Dialog.SetFocus
End Sub

Private Sub Command7_Click()
udp = udp + 1
If udp > 10 Then udp = 1
undo(udp) = Text1.Text

Text1.Text = "<HTML>" + cr + "<head>" + "<meta http-equiv=" + Chr(&H22) + "Content-Type" + Chr(&H22) + " content=" + Chr(&H22) + "text/html; charset=unicode" + Chr(&H22) + ">" + cr + "</head>" + cr + "<body>" + cr + cr + "</body>" + cr + "</HTML>"
End Sub

Private Sub DHTMLEdit1_DocumentComplete()
Text1.Text = DHTMLEdit1.DocumentHTML
If lof = 1 Then
undo(0) = Text1.Text
undo(1) = undo(0)
For i = 2 To 10
  undo(i) = ""
Next
udp = 1: lof = 0
End If
End Sub

Private Sub Form_Load()
cr = Chr(13) + Chr(10)
url = "http://www.vwalker.com/"
lof = 1
DHTMLEdit1.LoadURL (url)
baseurl = DHTMLEdit1.baseurl
Text2.Text = url
End Sub

Private Sub Command4_Click()
Dim l As Long
Dim st1 As String, st2 As String, st3 As String
Dim c As String
Dim flg As Integer
Dim lf As String

st2 = ""
c = Chr(13)
lf = Chr(10)
l = Len(Text1.Text)
For i = 1 To l
st1 = Mid(Text1.Text, i, 1)
If flg = 0 Then
  If st1 = lf Then st1 = cr
End If
If st1 = c Then flg = 1 Else flg = 0
  st2 = st2 + st1
Next
Text1.Text = st2

End Sub

Private Sub Form_Resize()
Dim h1, h2, v1, v2, h0, v0
If Form1.Width < 2000 Then Form1.Width = 2000
If Form1.Height < 2000 Then Form1.Height = 2000

h0 = Form1.Width
v0 = Form1.Height

Debug.Print h0

If ((h0 < 12000) And (h0 < v0 * 2)) Or ((h0 > 11999) And (h0 < v0 * 1.3)) Then
 v0 = v0 - 300 - 400
 v1 = v0 * 0.6
 v2 = v0 * 0.4
 DHTMLEdit1.Width = h0 - 80
 DHTMLEdit1.Height = v1
 Text1.Left = 0
 Text1.Top = v1 + 300
 Text1.Height = v2
 Text1.Width = h0 - 80
 Command1.Caption = " ↓ "
 Command2.Caption = " ↑ "

Else
 v1 = v0 - 700
 DHTMLEdit1.Width = h0 / 2 - 80
 DHTMLEdit1.Height = v1
 Text1.Top = 0
 Text1.Left = h0 / 2
 Text1.Height = v1
 Text1.Width = h0 / 2 - 80
 Command1.Caption = "→"
 Command2.Caption = "←"

End If

Command1.Top = v1 + 5
Command2.Top = v1 + 5
Command3.Top = v1 + 5
Command4.Top = v1 + 5
Command5.Top = v1 + 5
Command6.Top = v1 + 5
Command7.Top = v1 + 5
Text2.Top = v1 + 5
Text2.Left = 600
Command3.Left = h0 - 740 - 100
Command2.Left = h0 - 740 * 2 - 100
Command1.Left = h0 - 740 * 3 - 100
Command4.Left = h0 - 740 * 3 - 100 - 1100
Command5.Left = h0 - 740 * 4 - 100 - 1100
Command6.Left = h0 - 740 * 5 - 100 - 1100

End Sub

Private Sub Text2_KeyDown(KeyCode As Integer, Shift As Integer)
If KeyCode = 13 Then
lof = 1
DHTMLEdit1.LoadURL (Text2.Text)
baseurl = DHTMLEdit1.baseurl
End If
End Sub

Public Sub rtext(mode As Integer)

ss1 = Dialog.Text1.Text
ss2 = Dialog.Text2.Text
rs = ""
hs = Text1.Text

ls = Len(ss1)
lr = Len(ss2)

udp = udp + 1
If udp > 10 Then udp = 1
undo(udp) = Text1.Text
sp1 = 1
tp1 = 1
If mode = 0 Then
 sp0 = sp1
 sp1 = InStr(1, hs, ss1)
 If sp1 > Len(hs) Then sp1 = 0

Do Until sp1 < 1
 temp1 = Mid(hs, sp0, sp1 - sp0)
 Debug.Print sp1, temp1
 rs = rs + temp1 + ss2
 sp1 = sp1 + ls
 If sp1 > Len(hs) Then
   sp1 = 0
 Else
   sp0 = sp1
   sp1 = InStr(sp1, hs, ss1)
 End If
Loop
rs = rs + Right(hs, Len(hs) - sp0 + 1)
Text1.Text = rs
Else

End If

End Sub
```

という奴だ。これを読み書きすればコンポーネント内のHTMLを操作できそうだ。いろいろいじってみると，確かにこれでよさそうだし，意外と簡単に操作できる。ただし，書き換えたときはBaseURLもいじってやらないとちょっとおかしなことになる。それ以外はほとんど問題ない。

コンポーネントにWeb上のデータを読み込むには，LoadURLを使用する。指定するだけなので簡単。コンポーネントに表示されたページをマウスでいじってみると，それぞれのパーツごとにドラッグしての移動などができることがわかる。テキスト部分をクリックすればそのまま書き換えられる。機能としてはよい感じ。ただ，基本編集機能がキーボードショートカットのみというのは多少つらいかもしれない。コンテクストメニューは……まあ必要だったらつけるということで。

キーボードから多用されるのは，

ctrl-C　コピー
ctrl-X　切り取り
ctrl-V　張り付け
ctrl-Z　アンドゥ
ctrl-F　サーチ

あたりなので，普通に使っていればさほど迷うこともないだろう。そうそう，もとがFrontPage系のものなので，入力時のリターンキーは<p>タグに変換される。改行はシフト＋リターンで行うということを覚えておこう。

HTMLの置換処理

これだけではあんまりなので，テキストの置換機能くらいはつけておこう。読み込まれたHTMLファイルは1個の文字列型変数にだらだらと読み込まれている。先頭から1文字ずつ比較して……なんてやってたら日が暮れそうなので(場合によると凄く遅い)，instr関数で一気に検索する。なお，こういう置換や並べ替えの基本だが，たとえば，

ABCDEFABCDEF

のCをGGGに置き換える場合，いちいちABGGGEFABCEFなどという文字列を作って処理を進めたりはしない。文字列内でデータの入れ替えや空きの詰め，文字列を伸ばすなどの処理を行うと効率が悪いので，別の変数に端から叩き込んでいくようにする。

今回は大文字/小文字のチェックや非チェックといった機能は入れてないので単純に処理している。では，大文字/小文字に関わりなく検索したい場合は，やはり1文字ずつ比較していくしかないのだろうか？　そんなことはない。そういう場合は，前処理で全部大文字に変換した文字列を用意しておき，それに対して検索を行うか，あるいはサーチ文字列側で，大文字/小文字の組み合わせを総当たりで検索を行うなりしたほうが効率はよい。後者の方法は文字列長にもよるのだが，「大文字のみ」「小文字のみ」「キャピタライズ」くらいの可能性しかないと断定できるシステムであれば3回サーチのほうがたいていの処理より高速で単純だろう(それくらいBASICでやる処理は遅い)。

なんやかんやで，テキストボックス内のデータを比較するので，改行などを含んだ文字列の置換ができ，それなりに使いでのある処理ができあがった。

なお，いろいろやってたらCD-ROMには間に合わなくなったので，

http://www.vwalker.com/publishing/OhX/webx/

でプログラムをダウンロードしてほしい。

問題点

ざっとテキスト側からもGUI側からもいじれるようにはなったのだが，GUI側でいじったソースを取り込むと，勝手に整形(ぐちゃぐちゃにするといったほうが適切か)してくれるので非常に使いづらい。さらに，マイクロソフト流の整形規則は見境なしに全体に対して行われるので，JavaScriptなどが動かなくなることがある(変なところにスペースを入れられたり改行されたりする)。うーむ。FrontPageなら整形禁止を指定すると少しはマトモになるのだが，こいつにはそういう指定は不可能だ(一応レジストリとかも覗いてはみた)。

いくつか規則性はある。タグの前後以外のテキスト中の改行は無効で，テキストは全部つないで出力する。その際に，途中にあった改行はそのテキストの直前にまとめて吐き出す。また，英文の途中に半角空きがあるとそこで改行される傾向が強い。ソース段階でいじれるのでどうにでもなることは確かなのだから，それらを見据えて修正ないしPrettyPrintしてやればよいのだろうが……。ちょっと今後の課題として置いておこう。

あとは，表示ウィンドウが横長になると自動的にウィンドウ内を横分割から縦分割に変更するなど，使い勝手に気を遣ったつもりだが実際使ってみると，つくづく余計な機能だったなあと反省している。削るのはすぐできるのだけど参考までにそのまま残してある(せっかく作ったんだし)。

最後に

いろいろ解説してきたが「なるほど，Vwalkerはそういう風にしてやってるのか」と思い込むのは早計だ。プロの現場をなめちゃいけない。見たところ彼らは基本的に手作業である。「そういう定型の作業はこれを使えば3倍は効率上がります」的なツールを作ってあげてもタグの手打ちがお好みのようだ(タグをちゃんと理解しているとも思えないのだが)。編集者というのもなかなか頑固な生きものである。

表1　DHTMLEditのプロパティ，メソッドなど

種別	名称	型	読み書き	備考
Property	AbsoluteDropMode	Boolean		Property AbsoluteDropMode
Property	ActivateActiveXControls	Boolean		
Property	ActivateApplets	Boolean		Property ActivateApplets
Property	ActivateDTCs	Boolean		Property ActivateDTCs
Property	Appearance	DHTMLEDITAPPEARANCE		Property Appearance
Property	BaseURL	String		Property BaseURL
Property	BrowseMode	Boolean		Property BrowseMode
Property	Busy	Boolean	読み取り専用	Property Busy
Property	CausesValidation	Boolean		コントロールがフォーカスを失ったときに入力検査を行うかどうかを設定します。値の取得も可能です
Property	Container	Object		オブジェクトのコンテナを返します
Event	ContextMenuAction(itemIndex, Long)			
Property	CurrentDocumentPath	String	読み取り専用	Property CurrentDocumentPath
Event	DisplayChanged()			
Event	DocumentComplete()			

Property	DocumentHTML	String		Property DocumentHTML
Property	DocumentTitle	String	読み取り専用	Property DocumentTitle
Property	DOM	IHTMLDocument2	読み取り専用	Property DocumentObjectModel
Sub	Drag([Action])			ライン，メニュー，シェイプ，タイマの各コントロール以外のオブジェクトのドラッグ操作を，開始，終了，またはキャンセルします
Event	DragDrop(Source, Control, X, Single, Y, Single)			ドラッグ＆ドロップ操作が完了したときに発生します
Property	DragIcon	StdPicture		ドラッグ＆ドロップ操作中に，ポインタとして表示されるアイコンを設定します。値の取得も可能です
Property	DragMode	Integer		ドラッグモードで手動と自動のどちらを指定するかを設定します。値の取得も可能です
Event	DragOver(Source, Control, X, Single, Y, Single, State, Integer)			ドラッグ＆ドロップ操作が実行されているときに発生します
Function	ExecCommand(cmdID, DHTMLEDITCMDID, [cmdexecopt, OLECMDEXECOPT=OLECMDEXECOPT_DODEFAULT], [pInVar])			method ExecCommand
Function	FilterSourceCode(sourceCodeIn, String)	String		method FilterSourceCode
Event	GotFocus()			オブジェクトがフォーカスを受け取ったときに発生します
Property	Height	Single		オブジェクトの外観の高さを設定します。値の取得も可能です
Property	HelpContextID	Long		既定のヘルプファイルのコンテキスト番号をオブジェクトに指定します
Property	Index	Integer	読み取り専用	コントロール配列内のコントロールを特定する番号を設定します。値の取得も可能です
Property	IsDirty	Boolean	読み取り専用	Property IsDirty
Property	Left	Single		オブジェクトの内側の左端とコンテナの左端の間隔を設定します。値の取得も可能です
Sub	LoadDocument(pathIn, [promptUser])			method LoadDocument
Sub	LoadURL(url, String)			method LoadURL
Event	LostFocus()			オブジェクトがフォーカスを失ったときに発生します
Sub	Move(Left, Single, [Top], [Width], [Height])			オブジェクトを移動します
Property	Name	String	読み取り専用	プログラムで使用するオブジェクトを識別するための名前を返します
Sub	NewDocument()			method NewDocument
Property	Object	Object	読み取り専用	コントロール内のオブジェクトを返します
Event	onblur()			
Event	onclick()			
Event	ondblclick()			
Event	onkeydown()			
Event	onkeypress()			
Event	onkeyup()			
Event	onmousedown()			
Event	onmousemove()			
Event	onmouseout()			
Event	onmouseover()			
Event	onmouseup()			
Event	onreadystatechange()			
Property	Parent	Object	読み取り専用	オブジェクトが配置されているオブジェクトを返します
Sub	PrintDocument([withUI])			method PrintDocument
Function	QueryStatus(cmdID, DHTMLEDITCMDID)	DHTMLEDITCMDF		method QueryStatus
Sub	Refresh()			method Refresh
Sub	SaveDocument(pathIn, [promptUser])			method SaveDocument
Property	ScrollbarAppearance	DHTMLEDITAPPEARANCE		Property ScrollbarAppearance
Property	Scrollbars	Boolean		Property Scrollbars
Sub	SetContextMenu(menuStrings, menuStates)			method SetContextMenu
Sub	SetFocus()			指定したオブジェクトにフォーカスを移動します
Property	ShowBorders	Boolean		Property ShowBorders
Event	ShowContextMenu(xPos, Long, yPos, Long)			
Property	ShowDetails	Boolean		Property ShowDetails
Sub	ShowWhatsThis()			選択されたヘルプトピックをポップヒントの形式で表示します
Property	SnapToGrid	Boolean		Property SnapToGrid
Property	SnapToGridX	Long		Property SnapToGridX
Property	SnapToGridY	Long		Property SnapToGridY
Property	SourceCodePreservation	Boolean		Property SourceCodePreservation
Property	TabIndex	Integer		親フォーム内での，オブジェクトのタブオーダーを設定します。値の取得も可能です
Property	TabStop	Boolean		Tabキーを使ってフォーカスを移動するとき，そのオブジェクトにフォーカスが移動するかどうかを示す値を設定します。値の取得も可能です
Property	Tag	String		プログラムから参照するさまざまなデータを格納します
Property	ToolTipText	String		マウスポインタをコントロール上にあわせた場合に表示される文字を設定します。値の取得も可能です
Property	Top	Single		オブジェクトの内側の上端とコンテナの上端の間隔を設定します。値の取得も可能です
Property	UseDivOnCarriageReturn	Boolean		Property UseDivOnCarriageReturn
Event	Validate(Cancel, Boolean)			入力検査を行うコントロールがフォーカルを失うときに発生します
Property	Visible	Boolean		オブジェクトを表示するか非表示にするかを指定する値を設定します。値の取得も可能です
Property	WhatsThisHelpID	Long		オブジェクトに関連するコンテキスト番号を設定します。値の取得も可能です
Property	Width	Single		オブジェクトの幅を設定します。値の取得も可能です
Sub	ZOrder([Position])			指定されたオブジェクトの同一階層内でのZオーダーを変更します

いろいろなCGIの呼び出し＆値渡し方法
リンクを逆探知しよう

大和　哲　Yamato Satoshi

ブラウザがWebページにアクセスするときにはサーバといろいろなやりとりをします。そこで渡されるデータにはいろいろな情報が入っています。ここではPerl CGIを使って簡単なアクセス解析を行ってみましょう。あまり派手にやると嫌がられますが，結構面白いデータも取れるはずです。

「逆探知」などというと大袈裟ですけど，実はWebではページをアクセスさえしてくれれば「このページを見ている人がこの前にどこのページを見ていたか」を調べることは難しいことではなかったりします。というのも，Webブラウザは，どこかよそからのハイパーリンクをたどってそのページにきたときには，たいていリンク元ページをWebサーバには教えてくれているからです。CGIプログラムにはWebサーバから「HTTP_REFERER」という変数としてこのリンク元のURLがプログラムに渡されます。

ただ，たとえば，http://～.cgiというようにCGIプログラム自体が呼び出されるときに参照元を知る場合はこの方法で問題ないのですが，実際に利用することを考えると「実際にどこからリンクされているのだろう？」と知りたくなるのはこれから作るCGIページではなく，すでに存在するページであることが大多数だと思います。たとえば，index.htmlであったり，diary.htmlだったりするのではないでしょうか？

そこで，今回は「.htmlなどでも使える逆探知スクリプト」を目指してプログラムを書いてみることにしましょう。

プログラムの仕組みは簡単です。CGIで環境変数HTTP_REFERERを取ってURLを記録します。あまりデータが大きくなってしまうと困るので，ここでは仮に参照元のデータ（HTTP_REFERER）を見ることができた場合，直前のデータと同じでなければ最新の20個を記録しておくことにします。

記録するURL

ところで，この記録されるHTTP URLですが，HTTP_REFERERとしてCGIプログラムに渡されるデータは，単純にハイパーリンクでリンクされていたリンク元のURLが渡されます。で，そのためか，変なデータがときどき記録されることもあるようです。

たとえば，筆者のこれまでの経験では，

```
C:|bookmarks.html
```

などというURLが入っていたこともありました。(^^;)　おそらくハードディスク上によく行くWebページのURLを集めた「bookmarks.html」があって，それを表示させてから各ページを見てるのかもしれません。しかし，誰かのハードディスク上のファイルのように，インターネットから特定できない（見られない）ようなデータはあまり入っていても意味がないでしょうから，とりあえず，このプログラムではHTTP URLのみを記録することとして，先頭が「http://」でないものはカウントしないことにしておきます。

それから，HTTP URLの扱いなのですが，URLにはクエリー文字列やアンカー文字列がついている場合があります。クエリー文字列は，

```
http://www.foo.bar/test.cgi?message=ABCDE
```

のようにURLのパス名までが書かれたあとに「?」という文字といくつかの文字列が書かれます。

また，アンカー文字列は，

```
http://www.foo.bar/test.cgi#label1
```

という感じで，パス名（ディレクトリ＋ファイル名）までが書かれたあとに「#」という文字とアンカー名が書かれます。

クエリー文字列は，CGIなどにパラメータなどを渡すために使われます。アンカー文字列はその名前のアンカーをHTMLファイル中に埋め込んで，その名前の部分からブラウザで表示させるときなどに使われますね。

これらがURL中で書かれていると，そのデータがさまざまな用途に使われていることを示しています。

たとえば，クエリー文字列はInfoseekやgoogleなどの検索ページでは検索文字列が書かれているでしょうし，またページが更新されたかどうかを見にいく，いわゆるアンテナ系のCGIなどで使われているハイパーリンクのURLなどには，通常の.htmlで終わるURLの後ろに，更新を確認した時間などの情報がクエリー文字列として含まれていることがあります。また，アンカー文字列でも，それが当てになるデータである，という保証はありません。

ですので，これらをREFERERログを記録する際に，どのように扱うか

は，とても微妙なところです。取ったREFERERログをどのように使いたいかによって，このデータの扱いを決めることになるでしょう。

ほかにもいくつか問題はあります。同じページから飛んでくるのに，

http://foo.bar/diary.html#200009180500

と，

http://foo.bar/diary.html#200009182300

とでは実際には同じページから飛んできているわけですが，これを別物と思ってしまってよいのか，どうか，ですね。

今回のサンプルプログラムでは単純に過去20個のデータを記録するだけですので，強引に「別物だと思ってよいもの」と思って記録してしまうことにします。つまり，ロジックとしては，

先頭がhttp://であればURLとみなして記録する
記録するURLは＃や？も含めてすべて

ということにしてしまいます。

もし，これが単純なログ記録ではなく，たとえば，どこからいちばん多く参照されているのか集計したい，などという場合には，このあたりはきちんと考えてどれとどれを同一とみなすかを考えて実装しないと変なことになってしまうでしょうから，気をつけてくださいね。

CGIで作ってみた場合

さて，ということで，まずは最初の一歩として作ってみたのが「CGIそのものが呼び出されるとREFERERがログに記録されるプログラム」。サンプルリスト1，2です。

Webサーバ上には，CGIが実行できるディレクトリにリスト1，2と，

refererlog.dat

という名前でサイズ0，パーミションが666のデータを置く必要があります。

図1　自分の日記ページに貼り付けてみました。こんな感じ

このファイルがHTTP_REFERERデータの記録されたログファイルになるわけですね。

機能をもう少し詳しく紹介すると，まずリスト1のプログラムは呼び出されるとクエリー文字列からHTTP_REFERER変数を取り出します。そこに書かれているURLを取り出して，もし，

ログファイルの先頭に書かれているものと同じでない
「http://」で始まる
自分のWebページのURLは含まれない

だった場合，ログファイルの先頭にREFERERデータを追加して，もし最高個数である20個よりも多くなる場合は20個までを書き出すようにしています。それから，CGIの出力結果としては「banner.gif」というGIFファイ

注　検索ページのデータを解析してみよう

今回は単純に過去10個の参照元を表示するために，参照元のページのURLの「?」以降も単純に記録してしまっていますが，実は検索ページの「?」以降をまじめに解析するとかなり面白かったりします。というのも，Infoseekやgoogleといった検索ページの多くでは，参照元のURLの「?」以降に検索に使った文字列がURLエンコードしてそのまま残されているからです。たとえば，

http://www.infoseek.co.jp/Titles?qt=%C2%E7%CF%C2%C5%AF&lk=noframes&svx=100600&col=JW

これの，

qt=%C2%E7%CF%C2%C5%AF

の部分はデコードすると，

「大和哲」

になります。

名前などはまぁ，ありがちなんですが，こちらなどは例外で，むしろ「お，こんな文字列でもうちのページが引っかかるのか」と感心するような単語で検索してくる人が多いので，眺めているだけでも面白いのです。

たとえば，筆者はWeb日記ページを書いているのですが，この過去アーカイブ（http://web.pe.to/~deyamato/old00.html）に，この1週間で跡が残されていた検索語はこんな感じでした。

Crusoe 価格　値段
Oh!X 発売日 2000年 秋号
コスプレイヤー
キャバクラ　新宿
sendmail 添付ファイル 切れる
ロイヤル　銀座　バニー
アセンブラ 命令 一覧 Z80
吉四六　居酒屋　焼酎

……これだけ見ていると，なんのページじゃ，という感じがしますね，実に。

ちなみに今回のサンプルプログラムには特に単語をピックアップする機能はありませんがもし検索ページのURLが登録されていた場合，ビュアCGIからそのURLをクリックすると，その検索ページに飛ぶことができます。で，そのページを表示させて検索単語欄を見てみてください。いくつかのWebサイトではそこに使われた単語が表示されていることがあります。

ルをそのまま出力するようにしてあります。つまり，このCGIにリンク先から飛んでくると，Webブラウザにはぽつんとひとつ画像が表示されることになります(わぁ，さびし (^^;))。

ところで，自分のWebサイト内には当然，このCGIを呼び出すリンクも含まれるわけですが，自分のWebサイト内のデータは個人的には別にいいや，と思ったので最後の「自分のWebページのURLは含まれない」という条件が追加してあります。もし，これも見たいという場合は，この条件はプログラムリストからはずしてしまってください。

リスト2のほうはログビュアです。このCGIを実行すると単にURLが記録されているログファイルから，HTML文を生成して，URLにそのURLへのハイパーリンクを定義しています。つまり，平たくいうと，このCGIを実行して，ブラウザで見ると，

```
http://www.foo.bar/
http://www.yamato.nu/
http://web.pe.to/~deyamato
```

ブラウザにはこのようにURLが列挙されていて，このURLをクリックすると，書かれたURLにジャンプする，というわけですね。

ちなみに，このログブラウザに関しては，これから先の方法で収集しても同じように使えます。

既存のWebページで使うには

さて，このCGIなのですが，やはり面白くないですね。というのも，このCGI単独で置いていても，わざわざこのCGIの表示を目当てに飛んでくる場合にしか使えないからです。もし，これからコンテンツ自体も作る場合は，リスト1で出したようなGIFファイルを出力するのではなく，コンテンツのHTMLの中身をすべてここから出させればいいでしょうが，このような「逆探知」をしたい状況というのを考えると，これから作るのではなく，すでにあるコンテンツにこの逆探知プログラムを仕掛けておいて，どこからリンクがあるのか見たい，というのがありがちな状況でしょう。

が，CGIでない，すでにあるページでリンク元を参照したい場合はどうしたらいいでしょう。最初に思いつくのは，

```
<IMG SRC="linkcheck.cgi">
```

などのようにバナーやカウンタとして呼び出す方法でしょうね。でも残念，この方法だとページへのリンクではなく，このCGIへの呼び出し用のタグが書かれているページのURLがHTTP_REFERERに入ってしまい，たとえば毎回「index.html」というように同じファイルが記録されてしまうのです。これでは意味がないですよね。

SSIを使う

さて，そこで使える最初の方法はこれ。SSIを使う方法です。CGIの呼び出し方にはいくつかありますが，実は，SSIでページ中に<!--#exec cmdi="">で呼び出す場合は，サーバの1ページの処理の中に入るためなのか，HTTP_REFERERには，そのページにどこのページからリンクされているのかが書き込まれます。ですので，まずはこの方法を使ってリンク参照CGIを起動してみましょう。

SSIから呼び出される場合もCGIの作り方はまったく同じです。クエリー文字列のデコード方法なども普通にCGIを作った場合と同様にできますので，CGI用の汎用補助モジュールである，

```
cgi-lib.pl
```

などが利用できます。

SSIでは，cmd=で実行した場合，<!-- -->の部分を丸々書き換えるわけですから，そこに適当なHTMLコンテンツを出力するようにします。

たとえば，リスト1でGIFイメージ(rlogbanner.gif)をCGIから直に吐き出していましたが，今回は，

リスト1　REFERER記録

```
#!/usr/local/bin/perl

require "cgi-lib.pl";

#-------------------------------------
#REFERER LOGGER - その1(CGI版)
#       by de. 2000
#-------------------------------------

my $referer; #REFERER文字列が入る変数
my @queue;#URL20個の配列
my $myself="http://web.pe.to/~deyamato"; #省くアドレス
my $logfilename="refererlog.dat";

#画像を表示
open(GIF,"./banner.gif");
@gifdata = stat("./banner.gif");
$byte = $gifdata[7];
print "Content-type: image/gif¥n";
print "Content-length: $byte¥n¥n";
print <GIF>;
close(GIF);

#QUERY文字列からハッシュを作る
&ReadParse(*input);
@val = split(/&/,$input);
foreach $i(0 .. $#val){
		$val[$i] =~ s/%(..)/pack("c",hex($1))/ge;
		($name,$value) = split(/=/,$val[$i],2);
		$value =~ s/¥+/ /g;
		$val{$name} = $value;
}

if($ENV{HTTP_REFERER} ne ""){
		$referer = $ENV{HTTP_REFERER};
}

if($referer =~ /^http:¥/¥//){
		if(!($referer =~ /$myself/)){
		Getlogfile($referer);
		}
}

#終わり処理
close(ERRLOG);
exit(0);

sub Getlogfile{

		($current_url)=@_; #現在のアクセス元URL
		my $last_url; #過去の中で最新のURL

		my $date,count;
		my $count=0;

		open(LOG,$logfilename)|| print ERRLOG "LOGFILE READ ERROR¥n";
		while(<LOG>){
		chop;
		&put_queue($_);
		$last_url=$_;
		}
		close LOG;

		#--ログのURLをチェック
		if($current_url ne $last_url){
		&put_queue($current_url);
		}

		#-- ログファイルに書き出す
		open(LOG,">$logfilename")|| print ERRLOG "LOGFILE WRITE ERROR¥n";
		for($i=20;$i>=0;--$i){
		if($queue[$i] ne ""){
		#もし，重複するURLは記録しないならコメントをはずす
		#if($queue[$i] ne $current_url){
		print LOG "$queue[$i]¥n";
		#}
		}
		}
		close(LOG);

}

sub put_queue{
		#print " ** put_queue!! **¥n";
		for($i=20;$i>0;$i=$i-1){
		$j=$i-1;
		#print "queue$j($queue[$j]) => queue$i($queue[$i])¥n";
		$queue[$i]=$queue[$i-1];
		}
		$queue[0]=$_[0];

}
```

```
<IMG SRC="./banner.gif">
```

という出力をして，同じディレクトリ上のbanner.gifをブラウザには表示させるようにしてみました。この出力の仕方も普通のCGIと同じ標準出力に書き出せばOKです。

GIFファイルをクリックするとリスト2のログビュア（rlogview.cgi）が見られるように，このSSIを組み込むHTMLファイルには，

```
<A HREF="rlogviewer.cgi">
<!--#exec cmd="/home/deyamato/ssilog.cgi" -->
</A>
```

とでもしておくと便利でしょうね。

そうそう，SSIでプログラムを呼び出す場合，ファイル名がサーバ上のフルパスでなくてはなりませんので気をつけてください。ということで作ってみたのがサンプルリスト3です。

リスト2　ログビュア

```
#!/usr/local/bin/perl

#REFERER Log Viewer
#                                        by de 2000

require "cgi-lib.pl";
require "jcode.pl";

&ReadParse(*input);

@val = split(/&/,$input);

foreach $i(0 .. $#val){
        $val[$i] =~ s/%(..)/pack("c",hex($1))/ge;
        ($name,$value) = split(/=/,$val[$i],2);
        $value =~ s/¥+/ /g;
        $val{$name} = $value;
}

print &PrintHeader;

print "<HTML><HEAD><TITLE>ログビューア</TITLE>¥n";
print "<META HTTP-EQUIV=¥"content-type¥" CONTENT=¥"text/html;charset=x-euc-
jp¥">¥n";
print "<LINK REL=stylesheet TYPE=¥"text/css¥" HREF=¥"pcbrowser.css¥">¥n";
print "<BODY BGCOLOR=¥"#fdfad2¥">¥n";
print "<H1>リンク探知・ログビューアー<BR></H1><H2>
[呼び出し元・最新20]<BR></H2><P>¥n";

open(SRC,"refererlog.dat");
while(<SRC>){
        chop;
        print "<A HREF=¥"";
        print "$_";
        print "¥">";
        print "$_";
        print "</A><BR>¥n";
}
print "</P><HR>¥n";
print "<DIV ALIGN=CENTER><A HREF=¥"index.html¥">
トップページへ</A><BR></DIV>¥n";

print "<HR></BODY></HTML>¥n";
```

リスト3

```
#!/usr/local/bin/perl

require "cgi-lib.pl";

#-----------------------------------
#REFERER LOGGER - その2（SSI編）
#       by de. 2000
#-----------------------------------

my $referer; #REFERER文字列が入る変数
my @queue;#URL20個の配列
my $myself="http://web.pe.to/~deyamato"; #省くアドレス
my $logfilename="refererlog.dat";

#画像を表示
#-- SSIからなので，HTMLをそのまま出力
print "<img src=¥"./banner.gif¥">";

（以下，リスト1と同じ）
```

JavaScriptを使う

さて，SSIを使ってREFERERを記録するスクリプトを作ってみたわけですが，SSIですと，CGI以上に使えないWebサーバも多かったりします。

ここで，もうひとつの方法を考えてみましょう。そもそもこのREFERER変数はブラウザが前に見たページを記憶していて，サーバに渡している変数です。ですので，実はSSIのようなサーバ側で実行されるのでなくても，クライアント側でスクリプトを実行することができれば，この変数をアクセスすることができるはずですね。Webブラウザ上で実行できるスクリプトといえばVBScriptあるいはJavaScriptです。

ここではJavaScriptを使ってREFERERを参照してサーバ上のログファイルにこのデータを書き込んでみることにします。

さて，JavaScriptの場合，Document.refererという変数にCGIでのHTTP_REFERERに相当する，リンク元のページのURLの文字列が入っています。これをどうにかしてWebサーバ上のログファイルに書き込めばいいのです。

ただ，JavaScriptには，直接サーバ上のファイルを書き換えるような機能はありません。セキュリティを考えれば，クライアント上で動くプログラムがほいほいサーバのデータを書き換えられるようではまずいですよね。CGIの場合はサーバ側でサーバが許した権限（たとえばnobodyであるとか，CGIのオーナーユーザーの権限であるとか）でサーバ上で動いていますので，サーバ管理者が行った設定さえしっかりしていればセキュリティホールになる可能性は低いのでサーバ上のデータをある一定範囲で書き換えることができるわけですが，（たとえば，CGIから見える範囲で，誰でも書き換えられるようなデータであれば書き換えができる。ログデータのパーミションを666にしたのはそのためです）Webサーバの管理者の手の届かないところから書き換えられるようになっているとまずい，ということです。

というわけで，サーバ上にあるプログラムにログデータを書き換えさせるようにするわけですが，この記事ではサーバ上のプログラムといえばCGIで

注　SSI

SSIとはServerSideIncludeの略です。HTMLファイル中に<!-- -->とコメント文のかたちでディレクティブ（指示）を書き込んでおくとWebサーバがそれを解釈し，それに応じて，このコメント部分を書き換えます。

代表的なディレクティブとしては，

```
<!--#flastmod file="～"-->
<!--#exec cgi="～"-->
```

などがあります。

ファイルを実行するには，

```
<!--#exec cmd="～"-->
<!--#exec cgi="～"-->
```

の2通りの方法があります。cmd=で呼び出した場合はシェルスクリプトであるとみなされて出力結果がそのままHTMLの一部になります。cgi=の場合は，MIMEヘッダが付加されるとみなされます（が，サーバによっては単純に最初の1行を飛ばして，あとはHTMLとみなすものがあるので注意）。

このSSIはサーバ側で実行されますので，お使いになるWebサーバがSSIを実行できるように設定されていなければなりません。セキュリティホールとなりやすいことからインターネットプロバイダやホスティングのWebサーバではCGIよりもさらに使える環境が少ないのが問題になるかもしれません。

す。:-)

ここではJavaScriptからサーバ上のCGIに値を渡して，サーバ上のCGIにHTTP_REFERERに相当するURLのデータをログに書き込ませることにします。

なお，当たり前ですが，この方法はブラウザがJavaScriptを有効にしている場合だけ利用できます。つまり，誰かが対象のページを見にきても，必ず足跡がつくわけではない，ということですね（SSI版もブラウザがREFERERを出してくれなければならないので，こちらも必ず足跡がつくわけではないのですが）。

JavaScriptからCGIに値を渡す

ということで，JavaScriptで取ったREFERER変数をサーバ側のCGIプログラムに渡す方法です。

その前にJavaScriptとはどんなものなのか，簡単におさらいしておきましょう。

SSIがサーバ側でHTMLコンテンツを書き換える仕組みであるのに対し，簡単にいうと，JavaScriptはクライアント側で実行されるスクリプトです。

その実行のされ方は（たとえば.jsファイルをHTMLとは別個に作って呼び出すなど）いくつか方法があるのですが，基本的にはHTMLから呼び出されたり，あるいはHTML中に埋め込まれたプログラムがブラウザがHTMLを表示する際に実行され，さまざまな効果を生み出します。

JavaScriptはHTML中に埋め込まれる場合，

```
<SCRIPT LANGUAGE = "JavaScript">
</SCRIPT>
```

という宣言の中にプログラムが書かれます。なお，通常は，JavaScript未対応のブラウザでこのタグが認識されず，このプログラム部分がそのまま表示されてしまうことになるために，

```
<SCRIPT LANGUAGE = "JavaScript">
<!--
（プログラム本体）
//-->
</SCRIPT>
```

のようにHTMLの注釈文を使って書かれます（「-->」はそのままですとJavaScriptの一部とみなされてしまいますので，JavaScriptにとって注釈となる「//」を使って，「//-->」のように書きます）。

で，JavaScriptからCGIに値を渡す方法ですが，JavaScriptでは「Document.write」命令などを使うと，あたかもそのスクリプトの書かれているHTMLテキスト内で文やタグが書かれているように振舞うことができます。たとえば，

```
<HTML>
<HEAD>
<TITLE>JavaScriptテスト</TITLE>
</HEAD>
<BODY>
<SCRIPT LANGUAGE = "JavaScript">
<!--
  Document.write （"テストテストテスト"）;
// -->
</SCRIPT>
</BODY>
</HTML>
```

というように書かれたHTMLテキストは，JavaScriptが実行可能なブラウザで表示させると，画面には，

テストテストテスト

という文字列が表示されます。つまり，

```
<HTML>
<HEAD>
<TITLE>JavaScriptテスト</TITLE>
</HEAD>
<BODY>
テストテストテスト
</BODY>
</HTML>
```

というHTMLファイルを読み込ませたのと同じ効果になるわけですね。

ですので，たとえば，この命令で，

```
<IMG SRC="http://www.foo.bar/rlog3.cgi?referer=
http://www.yamato.nu">
```

とでも表示させると，ブラウザはあたかも，この文字列がHTMLテキストにそのまま書かれていたかのようにCGIプログラムが起動し，クエリー文字列として，

```
referer=http://www.yamato.nu
```

という値が渡されるのです。もうわかりますね。このDocument.writeで，SSIからCGIを呼び出したのと同様，CGIを呼び出せばよいのです。その際にURLエンコーディングしたREFERERデータをクエリー文字列としてURLにつけてやれば，CGIにjavaスクリプトの確保したHTTP_REFERER相当データをCGIに渡すことができます。

JavaScript版サンプルプログラム

以上のような発想で作ったのがサンプルリストの4と5です。

リスト4はHTML中に埋め込まれるJavaScriptのサンプル。そして，5が呼び出されるCGIプログラムのサンプルです。リスト4を，（CGIプログラムのあるURLを正しく書き直してから）REFERERを記録したいページのHTML中に埋め込むことで，そのHTMLファイルがブラウザに読み込まれたときに，サーバ上のCGIが起動し，CGIプログラムがREFERERログをサーバ中のデータファイルに書き込むわけです。

ところで，すでにお気づきかもしれませんが，このJavaScript版では，CGIプログラムは必ずしもJavaScriptの書かれたHTMLファイルを同じWebサーバになければいけない，ということはありません。HTMLからCGIを呼び出すのに同じサーバになくてはいけないわけではない，というのと同じ理屈ですね。

ですので，もし，リンク元の逆探知をしたいページの置いてあるサーバがCGIを許可していなくても，ほかのCGIを許可しているサーバにこのCGIを置いて，HTMLからこのCGIを呼び出す，などという使い方もできます。

なお，このプログラムもログビュアはリスト2のプログラムで見ることができます（これはリスト5のCGIプログラムと同じサーバになくてはなりません）。リスト5のCGIの出力はリスト2のCGIへのハイパーリンクになっていますから，このプログラムでも「REFERER LOG採取中」のバナーをクリックすると，いままでの参照元URLがリストとなって表示されるわけですね。

CGI環境変数

HTTP_REFERER以外にも，WebサーバはCGI実行時にいろいろな環境変数を持っています。これを生かすといろいろ面白いことができるかもしれません。たとえばHTTP_USER_AGENT。このCGIをアクセスしたブラウザやWebロボットの名前ですね。これのログを取ってみたりすると世のなかにはいろいろなブラウザがあるのだなぁ，と感慨にふけることができ

るでしょう（WindowsのInternetExplorerなどだとレジストリの書き換えだけで，この情報は変えられるので，単に名前を変えてみているだけの人も多いのかもしれませんが）。

SERVER_SOFTWAREにはこのWebサーバの名前とバージョンが入っています。たとえば，「Apache/1.3.12 (Unix)」など。ほかの人へのサービスには使えるかどうかわかりませんが，自分の使っているWebサーバのバージョンを調べてみたりするのには使えますね。

REMOTE_HOSTはアクセスした端末のリモートホスト名（プロバイダのアクセスポイントの名前などになっていることが多いです），SERVER_ADMINでこのサーバの管理者のメールアドレスが入っていたりします。HTTP_COOKIEは有名なCockieですね。DATE_LOCAL（サーバのいる場所の現地時間）やREMOTE_USER（認証を利用している場合のユーザー名），TZ（タイムゾーン）なんていうのもありますね。プロクシを利用してアクセスしている場合はHTTP_X_FORWARDED_FOR（アクセスしているクライアントの本当のIPアドレス）などという変数を持つ場合もあります。

なお，Webサーバの設定によっては持たない境変数，あるいは特定の環境でだけ持つ環境変数なども存在します。また，サーバの種類（Apacheか，IISか，NetscapeServerか……など（によっても変化します。使っているサーバがどんな環境変数を出すか，試しにこんなCGIプログラムを使って一度確認してみるといいでしょう。

```
#!/usr/local/bin/perl

require "./cgi-lib.pl";
require "./jcode.pl";

&ReadParse(*input);
print &PrintHeader;
print'<META content="text/html; charset=charset=EUC-JP"
http-equiv=Content-Type>';
print &HtmlTop ("環境変数リスト");
print "<dl compact>";
foreach $key (sort keys %ENV) {
      print "<dt><b>$key</b></dt><dd>$ENV{$key}</dd>¥n";
}
print "</dl>";
print &HtmlBot;
exit(0);
```

図2　バナーをクリックするとログが表示されます。色違いがどこかで見たCGIと同じなのはお許しあれ（前回のスタイルシートをそのまま使ったもので）

参考文献
Apahe 日本語マニュアル
http: japache.infoscience.co.jp/

リスト4

```
<HTML>
<HEAD>
<META HTTP-EQUIV="content-type" CONTENT="text/html;charset=x-euc-jp">
<TITLE></TITLE>
<LINK REL=stylesheet TYPE="text/css" HREF="pcbrowser.css">
</HEAD>
<BODY BGCOLOR="#fdfad2">
<H1>このページについて<BR></H1><H2>[このページは…]<BR></H2><P>
このページは「Javaスクリプトと CGIの組み合わせで作る・リンク元探知CGIテストページ」です。<BR>
<DIV ALIGN=CENTER>
<A HREF="source.html">最初のページに戻る</A><BR>
</DIV></P>
(最終更新:<!--#flastmod file="index.html"-->)
<A HREF="rlogviewer.cgi">
<IMG src='rlogger.cgi?referer='>
</A>

<HR><BR>
<BR></P></BODY>
</HTML>
```

リスト5

```
#!/usr/local/bin/perl

require "cgi-lib.pl";

#------------------------------------
#REFERER LOGGER
#       by de. 2000
#------------------------------------

my $referer; #REFERER文字列が入る変数
my @queue;#URL11個の配列
my $myself="http://web.pe.to/~deyamato"; #省くアドレス
my $logfilename="refererlog.dat";

#画像を表示
open(GIF,"./banner.gif");
@gifdata = stat("./banner.gif");
$byte = $gifdata[7];
print "Content-type: image/gif¥n";
print "Content-length: $byte¥n¥n";
print <GIF>;
close(GIF);
##--

#QUERY文字列からハッシュを作る
&ReadParse(*input);
@val = split(/&/,$input);
foreach $i(0 .. $#val){
                    $val[$i] =~ s/%(..)/pack("c",hex($1))/ge;
                    ($name,$value) = split(/=/,$val[$i],2);
                    $value =~ s/¥+/ /g;
                    $val{$name} = $value;
}

#エラーログファイルを開く
open(ERRLOG,">rlog_err.log");

if(exists($val{"referer"})){
                    $referer=$val{"referer"};
}

if($referer =~ /^http:¥/¥//){
                    if(!($referer =~ /$myself/)){
                    Getlogfile($referer);
                    }
}

#終わり処理
close(ERRLOG);
exit(0);

sub Getlogfile{
(以下リスト2と同じ)
```

Tcl/Tkによるミニミニゲーム集 Part 3

広井 誠　Hiroi Makoto

Tcl/Tkによるゲーム集の第3弾です。多少慣れは必要ですが，比較的簡単にGUIアプリケーション環境を実現できるTcl/Tkは暇プロにもゲームにも適しており，マルチプラットフォームで使えるという点もポイントが高いシステムです。では新作ゲームをお楽しみください。

20世紀最後の年には時代が大きく変わることを強く実感させる出来事がいくつかありました。インプライズからBorland C++ Compiler 5.5無償で公開されたことにも驚きましたが，昔からのOh!X読者には，シャープからX68000用の基本ソフトがBIOSを含めてフリーで公開されたことと，満開製作所がX68000関連事業から撤退することのほうが大きな衝撃だったのではないでしょうか。筆者はOh!Xと同様に電脳倶楽部にも愛着を覚えていたので，Vol.147で休刊することは残念でなりません。時代の終焉を感じます。

一方，基本ソフトの公開により，X68000エミュレータEX68が本格的に利用されることになるでしょう。X68000に興味があっても実機が手に入らなかったり，さまざまな理由で実機を手放すことになったユーザーも，これからはWindowsでX68000を楽しむことができます。X68000にはフリーで利用できるツールや開発環境が整っているので，ゲームで遊ぶだけではなく，プログラミングを楽しむユーザーが増えることを期待しています。X68000の新しい時代は，EX68によって幕が開くのかもしれませんね。

また，Windowsでも最新のC/C++コンパイラを利用できるようになり，懐具合が気になるホビープログラマにはありがたいことです。Botchy氏が開発された DirectXを簡単に利用できるライブラリ，el(Easy Link Library)との組み合わせで，ゲームの制作も活発になるのではないでしょうか。Windowsではシェアウェアの割合が高いようですが，これからはフリーウェアが増えるかもしれません。

もっとも，公開されたコンパイラはDOS窓で利用するもので，統合開発環境ではありません。また，Windows用のクラスライブラリVisual ComponentLibrary(VCL)も付属していません。Windows用のアプリケーションを作るにはWindows APIを呼び出すことが必要になります。このため，初心者が手軽にGUIアプリケーションを開発できるわけではありません。

ホビープログラマが手を出せる範囲は確実に広がってきています。ですが，手軽にGUIベースのプログラミングを楽しむには，やっぱりTcl/Tkのようなスクリプト言語のほうが適しています。そこで，今回も2000春号に続いてTcl/Tkで簡単なゲームを作ってみました。

1　ならべてポン

ならべてポンは，同じ色のカードを左から順番に並べるパズルゲームです。このゲームでは，空いている場所にカードを移すことでゲームを進めていきます。カードを空き場所に移動させるには，次の条件を満たさなくていはいけません。

(1) いちばん左側の空き場所には1のカードを移すことができる
(2) それ以外の空き場所には，左隣のカードと同じ色で番号がひとつ大きいカードを移すことができる

たとえば，図1のゲーム開始時には(1，C)，(3，A)，(3，C)，(3，D)の4つの空き場所があります。(1，C)はいちばん左側なので，ここにy1，r1，b1，g1のどれかを移すことができます。(3，A)の左隣はy3ですが，これより大きなカードはないので，この場所にカードを移すことはできません。(3，C)の空き場所は，左隣のカードがb1なので(1，D)にあるb2のカードを移すことができます。

このようにカードを移動させて，左から順番にカードを並べます。どの行にどの色のカードを並べるかは自由です。この例ではA行に黄色を並べていますが，解けるのであればB，C，Dのどの行に並べてもかまいません。移動できるカードがなくなったならば「手詰まり」となります。

盤の大きさは，4行4列，5行5列，6行6列の3種類から選ぶことができます。盤が大きくなるとカードの種類と枚数が増えるのでパズルの難易度は高くなります。また，カードはランダムに配置されるので，解けない場合もあります。ご注意くださいませ。

2　SEVEN

SEVENは数字が書かれた72枚のカードをすべて取り除いていくパズルゲームです。カードは10行8列の盤に並べられていて，カードを取り除くには，次の条件を満たさなくてはいけません。

ゲーム開始時

	1	2	3	4
A	y1	y3		r3
B	r1	g2	g1	b3
C		b1		r2
D	b2	g3		y2

完成図

	1	2	3	4
A	y1	y2	y3	
B	g1	g2	g3	
C	r1	r2	r3	
D	b1	b2	b3	

y：黄，r：赤，b：青，g：緑

図1　ならべてポン

図2　ならべてポン

図3　SEVEN

(1) 同じ色のカードであること
(2) 同じ高さ(行)にあること
(3) 2枚のカードが足して7になること

カードを取り除いたあとの空き場所には，上のカードが落ちてきます。それから，数字以外にも「?」が表示されたワイルドカードが8枚あります。ワイルドカードをクリックし，次に数字カードをクリックすると，数字カードをワイルドカードの位置に移動させることができます。

スコアは数字カードをすべて取り除くまでの時間です。ただし，ワイルドカードを使うと，1枚につきスコアが10秒加算されます。ワイルドカードをうまく使って，ハイスコアを目指してください。

3 ブロックアップ

ブロックアップは，落ち物系パズルを逆にしたようなゲームで，下からブロックを押し上げて，積まれているブロックを消していくパズルゲームです。一番下のラインにある2個ひと組のブロックを左右に動かし，適当な位置でブロックを押し上げてください。ブロックには色がついていて，同じ色のブロックを縦，横，斜めのいずれかに3つ以上並べると，そのブロックを消すことができます。ブロックは複数の方向に重複して並んでいてもかまいません。消したあとの空間には上のブロックが落ちてきます。

ブロックの操作はキーボードを使って行います。

4，カーソルキー左　：ブロックを左に移動
6，カーソルキー右　：ブロックを右に移動
5，カーソルキー下　：ブロックの順番を入れ替える
8，カーソルキー上，スペース：ブロックを押し上げる
S　：ゲームの開始

ブロックは6，7，8種類の3つのなかから選ぶことができます。ブロックの種類が多くなるほどゲームの難易度は高くなります。得点は，

(消したブロックの数 ＊ 連鎖ボーナス) の2乗

です。連鎖とは，ブロックが消えて上から落ちてきたブロックによって，同じ色のブロックが3つ以上並んでブロックが消えることです。連鎖ボーナスは連鎖回数が増えるとともに1→2→4→8→16……と増加していきます。時間制限はありません。うまく連鎖を起こして高得点を狙ってください。

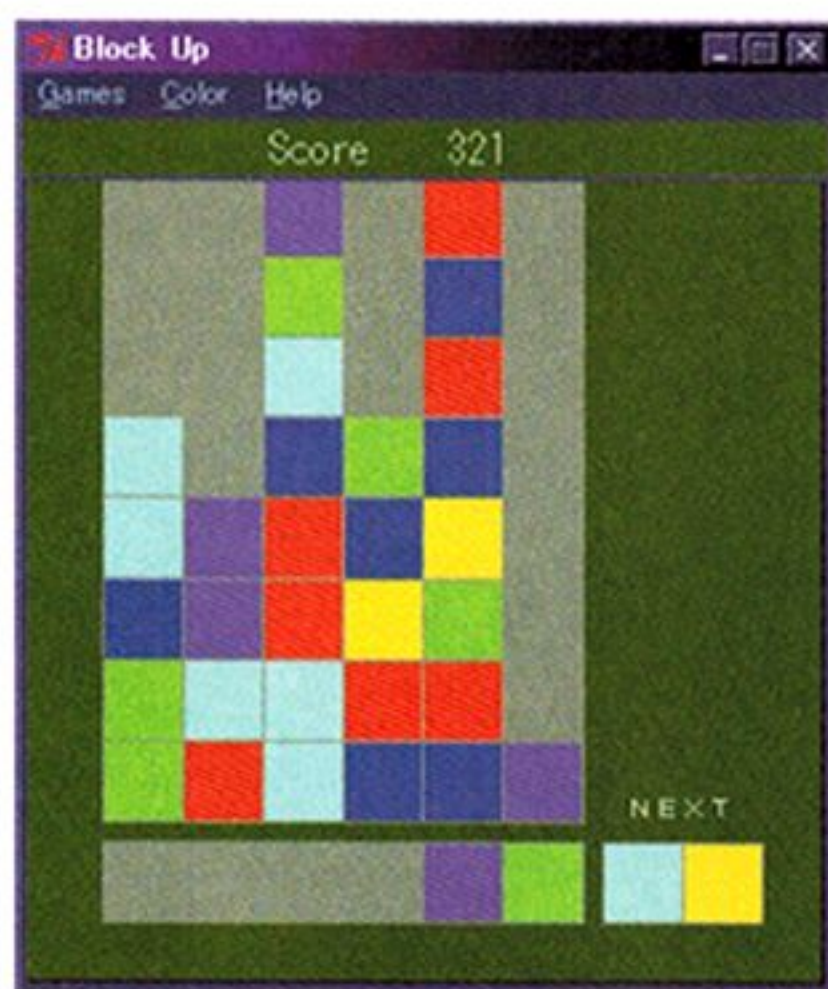



図4 ブロックアップ

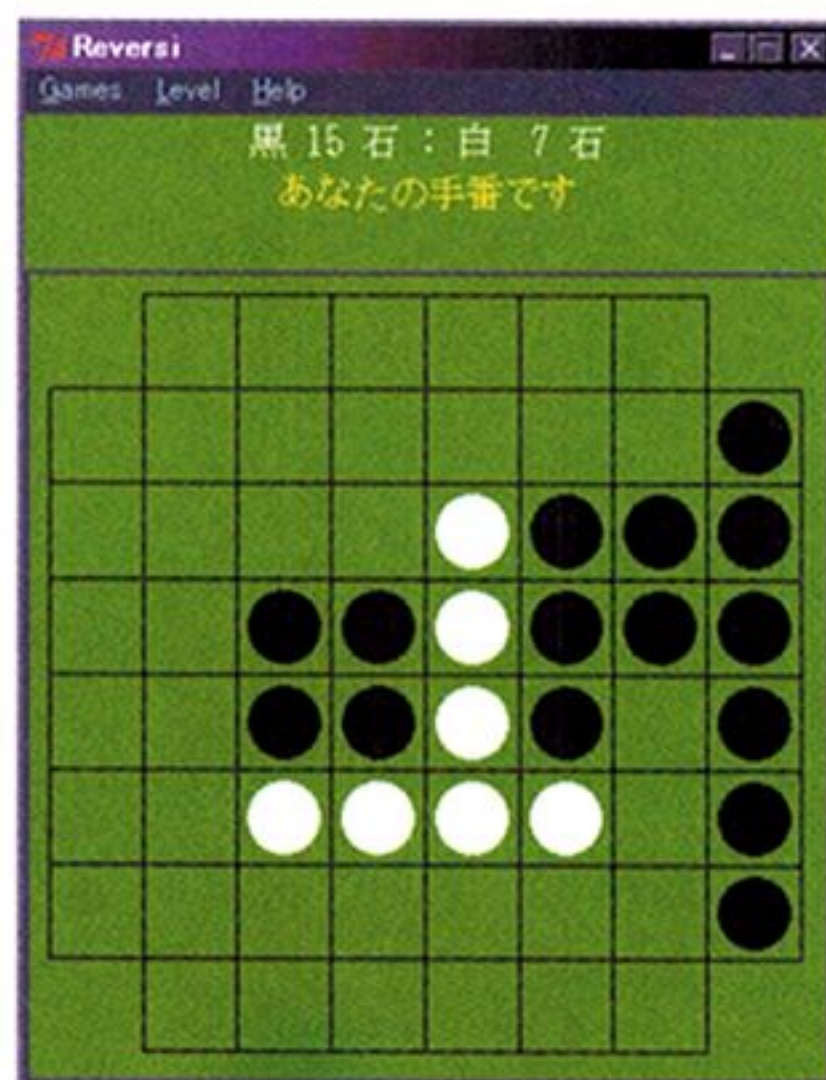



図5 OCT REVERSI (オクトリバーシ)

4 OCT REVERSI(オクトリバーシ)

思考ゲームの定番「REVERSI(リバーシ)」の変形バージョンです。通常のリバーシは盤面が8行8列ですが，オクトリバーシでは8行8列盤の4カ所の隅を取り除いた盤面を使用します。このため，石の総数は60個と少なくなりますが，逆に隅は8カ所と増えることになります。ルールはリバーシと同じです。

コンピュータの強さは，レベル0から5までの6段階です。

Level 0　：1手読み，残り 8，9手で読み切る
Level 1　：2手読み，残り 9，10手で読み切る
Level 2　：3手読み，残り10，11手で読み切る
Level 3　：4手読み，残り11，12手で読み切る
Level 4　：5手読み，残り12，13手で読み切る
Level 5　：6手読み，残り13，14手で読み切る

レベルを上げるとコンピュータは強くなりますが，その分考える時間が長くなります。特に，残り13，14手で最後まで読み切る場合，少々時間がかかることがあります。高速CPUを搭載したパソコンを使ったほうが快適にプレイできます。

思考ルーチンは，基本的にはリバーシと同じですが，評価関数の調整が十分ではありません。このため，慣れると簡単に勝てるようになるだろうと思っていました。ところが，実際にプレイしてみると，レベルが高くなるとなかなか勝つことができません。まあ，4石少ない分だけ読み切りモードに入るのも早くなるのですから，コンピュータには有利なゲームなのかもしれません。また，ゲーム感覚がリバーシとヘックスリバーシの違いよりも大きく，最初は相当に戸惑うと思います。リバーシとの違いをお楽しみください。

このゲームはTcl/Tk 8.2を使って作られています。オクトリバーシの思考ルーチン用DLLは，バージョン8.2以前のTcl/Tkではロードすることができません。オクトリバーシを実行する場合，Tcl/Tk 8.2をインストールしてください。インストール後，Tclスクリプトファイルをダブルクリックするとゲームが実行されます。また，スコアを記録するゲームがあるため，ハードディスクなど書き込み可能なメディア上で実行してください。

おわりに

Tcl/Tkは手軽にGUIアプリケーションを作成できる優れた開発環境です。筆者のようにTcl/Tkでゲームを作るユーザーは珍しいようですが，ゲーム以外のGUIアプリケーションも簡単に作ることができます。驚くほど簡単にプログラムを作ることができるので，興味のある方はぜひTcl/Tkを使ってみてください。そして，便利なアプリケーションができたらOh!Xに投稿してくださいね。

権利・免責事項など

このプログラムはフリーウェアとします。自由に使ってください。ただし，このプログラムは無保証であり，使用したことにより生じた損害について，作者は一切の責任を負いません。また，このプログラムを販売することで利益を得るといった商行為は禁止します。

参考文献

[1]『入門 tcl/tk』久野靖，アスキー出版局，1997
[2]『Tcl&Tkツールキット』John K. Ousterhout，西中芳幸 石曽根信 訳，ソフトバンク，1995
[3]『Tcl/TkによるWindows GUIプログラミング』須栗歩人，C MAGAZINE 1998年11月号，ソフトバンク
[4]『Effective Tcl/Tk』Mark Harrison/Michael McLennan，吉川邦夫訳，アスキー，1999

新DirectX Retained Mode講座　3次元でポン！(前編)
3Dの基本を押さえよう

飯田 伸一　Iida Shinichi

DirectX8時代のRetainedMode入門です。これまでのRetainedMode入門はひとまず忘れてこれからDirectXを始めようという人のためにプログラム作成の手順を追って解説していきます。

3次元の世界で捕まえて "Beyond the fanatic world"

唐突だが諸般の事情により，前任者である菊地氏から直接のご指名を受け，DirectX Retained Mode（以下，RM）講座を引き継がせていただくことになった。*1「好きにやって構わない」ということだったので内容的にどのようなものにしようかといろいろと迷ったのが正直なところだが，僕なりの経験・解釈によるRM再入門のようなものを展開していこうと思う（菊地氏のRM講座で基本を学ばれた読者には申しわけないが）。……C（C++）をひととおりは理解できるようになり，3Dという世界に踏み込みたいが，なにやら難しそうで躊躇したり，高価な書籍をいろいろ買ってはみたが，よくわからなくて途方に暮れてしまっている人……。まさに3次元というフィールドの縁石に立っている，そういった人たちが転落しないように，僕の記事がいわゆる「3次元世界の捕手」の役割になってくれればいい。そんな気持ちを込めて，原稿を書きたい。

＊1　前任者である菊地氏のような，あらゆる意味で高度な記事内容を期待していると，ほぼ確実に足元をすくわれるので注意。自分の身のほど知らずは，百も承知，二百も合点なんである。

DirectX概説およびRMを使用することの有効性について

DirectX（以下DX）とは，現在Windows上でゲームを制作するのにもっともよく利用されているAPIだ。DXではアプリケーション開発を支援するために，さまざまな機能を提供しており，描画関係ではスプライトなどの2D描画が目的のDirectDraw，3Dを処理するDirect3Dが用意されていた。Direct3Dには，比較的簡単に3Dを使ったアプリケーションが作れるRM（保持モード）と，難解だが扱い方に柔軟性があるIM（直接モード）の2つがあった……。

が，しかしDX7から登場したヘルパーライブラリ（DX3DX）によって，IMが格段に扱いやすくなり，プログラマ側の負担が大幅に減った。さらにDX8に至っては，ほぼ完全に2D（DirectDraw）と3D（Direct3D）の融合が図られている。*2

その結果としてRMは，もはや過去の遺物として取り残された感もあるのだが，まだまだ3Dを学習する最初の入り口としての有用性はある。というのも，すでに3Dの基本がわかっていて，IMをバリバリと使っているハッピーな人は別として，これからゲームなどを作ろうとしている初心者には，IMはまだまだ敷居が高いのだ（IM自体，プロ開発者向けなのだから，ある意味ではしかたがないのだが）。

DX3DXによってあらゆる手続きが簡略化されたとはいえ，IMを扱うには，ちょっとしたお勉強が必要だし，それに初心者にとっては3Dプログラミング以前に，いろいろと知っておかなければならないことも多い。これでは実際に開発の現場で使用している方や，将来プロを目指すというキアイの入っている方以外は，手軽にゲームを作ろうとすると確実に挫折してしまうだろう。……それは面白くない。

確かにRMはDX3DXと比べると，あちこち初期化したり面倒だと感じるかもしれない。が，学習する側に配慮して，本質的なことに焦点を置いて，できるだけ簡潔に解説をするつもりなので，当講座においては，読者の皆様にはそんなに負担はかからないはずだ。

本格的にIMに移る前に，ワンクッション置くという意味で，まだRMは有用性を失ってはいないと思うのだ。

＊2　はっきりとDX7のオンラインヘルプで，今後のサポートは保証しないと明記されている。

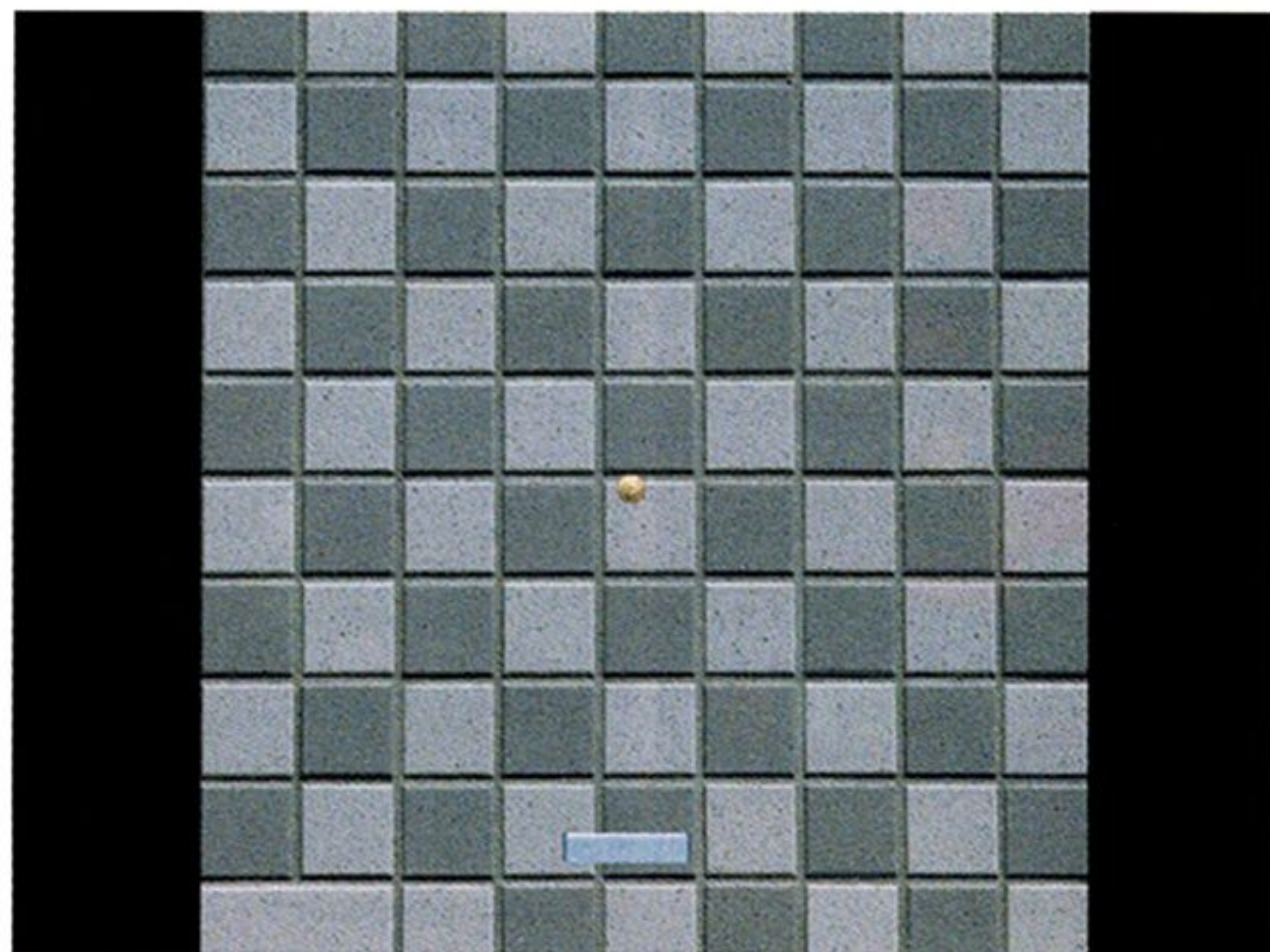
COLUMN 1　DirectXでプログラムを作るまで

この講座で出てくるサンプルプログラムをコンパイルしたり，自分で一から始めてアプリケーションを開発したい場合には，DirectX SDKをインストールしたあとに，利用しているコンパイラの設定が必要となる。この設定をしないと，ほぼ確実にエラーが出るので，なにはともあれ最初にDirectX SDKの各種ライブラリをコンパイラへ設定しなければならない。

まず，メニューバーの[ツール]－[オプション]－[ディレクトリ]タブを選び，ディレクトリを設定する。

[表示するディレクトリ]を[インクルードファイル]にして，たとえばSDKが"MSSDK"というフォルダに入っているとすると，C¥MSSDK¥INCLUDEを加える。

次に[ライブラリファイル]で，C¥MSSDK¥LIBを追加する。なお，これらのインクルードファイルやライブラリファイルのディレクトリ設定は，両方ともリストのいちばん先頭に持っていくこと。これで，サンプルプログラムがビルド可能になる。

自分で新規プロジェクトを組みたい場合には，[新規作成]－[プロジェクト]－[Win32 Application]を選んで，「単純なWin32アプリケーション」を選択する。あとは，[プロジェクト]－[設定]で，[リンク]タブを指定し，いちばん下のプロジェクトオプション欄の先頭に"dxguid.lib ddraw.lib d3drm.lib"(これは最低限の構成なので，各自必要に応じて適宜，追加していってほしい)を上書きする

画面1 ごく単純な2Dのピンポンゲーム

リスト1

```
/ DirectDraw サンプルゲーム
// PingPong.cpp

#include "stdafx.h"
#include "pingpong.h"

#define APPNAME "PingPong"

IDirectDraw *lpDD=NULL;                  // DirectDrawオブジェクト
IDirectDrawSurface *lpDDPrimary=NULL; // プライマリサーフェス
IDirectDrawSurface *lpDDBack=NULL;    // バックサーフェス
IDirectDrawSurface *lpDDPad=NULL;     // オフスクリーンサーフェイス
IDirectDrawSurface *lpDDBall=NULL;    //       "
IDirectDrawSurface *lpDDBG=NULL;      //       "

HINSTANCE hAppInst;
LRESULT CALLBACK WndProc(HWND, UINT, UINT, LONG);

BOOL bActive = FALSE;
DWORD lastCount;
// ボール
struct ball{
  float X,Y;      // 座標
  float XAdd,YAdd; // 移動量
}Ball;
// パドル
struct pad{
  int X;      // 描画座標
  int XAdd;  // 移動量
}Pad;
// 反発力テーブル
float PadDir[16][2]={
    -1.99036944f,-0.19603428f,  //左
    -1.71388071f,-0.92603428f,
    -1.71388071f,-0.98056933f,
    -1.76384258f,-0.94279349f,
    -1.54602087f,-1.26878655f,
    -1.26878655f,-1.54602087f,
    -0.94279349f,-1.76384258f,
    -0.58056933f,-1.91388071f,  //中央
    -0.19603428f,-1.99036944f,  //
     0.19603428f,-1.99036944f,
     0.58056933f,-1.91388071f,
     0.94279349f,-1.76384258f,
     1.26878655f,-1.54602087f,
     1.54602087f,-1.26878655f,
     1.76384258f,-0.94279349f,
     1.71388071f,-0.98056933f,  //右
};

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
  WNDCLASS wc;
  HWND hWnd;
  MSG msg;

  wc.style=CS_HREDRAW|CS_VREDRAW;
  wc.lpfnWndProc=WndProc;
  wc.cbClsExtra=0;
  wc.cbWndExtra=0;
  wc.hInstance=hInstance;
  wc.hIcon=LoadIcon(hInstance, IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground=(HBRUSH)GetStockBrush(BLACK_BRUSH);
  wc.lpszMenuName=0;
  wc.lpszClassName=APPNAME;

  RegisterClass(&wc);

  hAppInst=hInstance;
  // ウィンドウクラスの登録
  hWnd=CreateWindowEx(0, APPNAME, APPNAME, WS_POPUP, 0, 0,
    GetSystemMetrics(SM_CXSCREEN), GetSystemMetrics(SM_CYSCREEN),
    NULL, NULL, hInstance, NULL);

  if(!hWnd) return FALSE;

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);

  // アプリケーションの初期化
  if(!InitDDraw(hWnd)) return FALSE;

  // 初期化終了
  bActive = TRUE;

  // メッセージポンプ
  while(TRUE){
    if(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)){
      if(!GetMessage(&msg, NULL, 0, 0)){
        return msg.wParam;
      }
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
    if(bActive) Render();
  }
  return 0;
}

long FAR PASCAL WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  switch(message){

  case WM_ACTIVATEAPP:
    bActive=(BOOL)wParam;
    if(bActive && lpDD)
      ResumeScreen();
    break;
  case WM_CREATE:
    break;
  case WM_SETCURSOR:
    SetCursor(NULL);
    return TRUE;

  case WM_KEYDOWN:
    switch(wParam){
    case VK_ESCAPE:
      DestroyWindow(hWnd);
      break;
    }
    return TRUE;

  case WM_DESTROY:
    bActive=FALSE;
    ReleaseObjects();
    PostQuitMessage(0);
    break;
  }
  return DefWindowProc(hWnd,message,wParam,lParam);
}

// オブジェクトの開放
void ReleaseObjects()
{
  RELEASE(lpDDBG);
  RELEASE(lpDDBall);
  RELEASE(lpDDPad);
  RELEASE(lpDDBack);
  RELEASE(lpDDPrimary);
  RELEASE(lpDD);
}

// DirectDraw初期化エラー処理
BOOL InitDDError(HWND hWnd)
{
  ReleaseObjects();
  MessageBox
(hWnd, "Initialize Error", "DirectDraw Initialize", MB_OK|MB_ICONHAND);
  return FALSE;
}

// サーフェスの修復
BOOL ResumeScreen()
{
  DDCOLORKEY ddck;

  lpDDPrimary->Restore();
  lpDDBack->Restore();
  lpDDPad->Restore();
  lpDDBall->Restore();
  lpDDBG->Restore();

  HDC hDCDest;
  lpDDPad->GetDC(&hDCDest); // スプライトサーフェスのDCを取得
  // リソースからビットマップをロード
  HBITMAP hBmp=(HBITMAP)LoadImage(NULL, "Pad.Bmp", IMAGE_BITMAP,0, 0,
    LR_LOADFROMFILE|LR_CREATEDIBSECTION);
  HDC hDCSrc=CreateCompatibleDC(NULL);  // メモリデバイスコンテキストを作成

  // スプライトサーフェスへコピーする
  SelectObject(hDCSrc, hBmp); // ビットマップを選択
  BitBlt(hDCDest, 0, 0, 64, 16, hDCSrc, 0, 0, SRCCOPY); // コピー
  DeleteObject(hBmp);          // ビットマップを復帰
  lpDDPad->ReleaseDC(hDCDest);     // スプライトサーフェスのDCを開放

  lpDDBall->GetDC(&hDCDest);
  hBmp=(HBITMAP)LoadImage(NULL, "Ball.Bmp", IMAGE_BITMAP,0, 0,
    LR_LOADFROMFILE|LR_CREATEDIBSECTION );
  SelectObject(hDCSrc, hBmp);
  BitBlt(hDCDest, 0, 0, 16, 16, hDCSrc, 0, 0, SRCCOPY);
  DeleteObject(hBmp);
  lpDDBall->ReleaseDC(hDCDest);
```

```
  lpDDBG->GetDC(&hDCDest);
  hBmp=(HBITMAP)LoadImage( NULL, "BG.Bmp", IMAGE_BITMAP,0, 0,
    LR_LOADFROMFILE|LR_CREATEDIBSECTION );

  SelectObject(hDCSrc, hBmp);
  BitBlt(hDCDest, 0, 0, 640, 480, hDCSrc, 0, 0, SRCCOPY);
  DeleteObject(hBmp);
  lpDDBG->ReleaseDC(hDCDest);

  DeleteDC(hDCSrc); // メモリデバイスコンテキストを開放

  // カラーキーを設定
  ddck.dwColorSpaceLowValue  = 0; // 黒を指定
  ddck.dwColorSpaceHighValue = 0;
  lpDDPad->SetColorKey(DDCKEY_SRCBLT, &ddck);
  lpDDBall->SetColorKey(DDCKEY_SRCBLT, &ddck);

  return TRUE;
}

void Render()
{
  int x;
  RECT rect;

  rect.left=rect.top=0;
  rect.right=640;
  rect.bottom = 480;
  // 背景の描画
  lpDDBack->BltFast
  (0, 0, lpDDBG, &rect, DDBLTFAST_WAIT|DDBLTFAST_NOCOLORKEY);

  Pad.XAdd=0;

  //  キー入力
  if(GetAsyncKeyState(VK_RIGHT))  Pad.XAdd=10;
  if(GetAsyncKeyState(VK_LEFT)) Pad.XAdd=-10;

  if(Pad.X>525-42) Pad.X=525-42;
  if(Pad.X<95) Pad.X=95;

  rect.right=64;
  rect.bottom=16;
  // パドルの描画
  lpDDBack->BltFast(Pad.X, 416, lpDDPad,&rect,
    DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);

  Pad.X+=Pad.XAdd;  //  パドルの移動

  // パドルとボールの当り判定
  if((int)Ball.Y>416-16 && (int)Ball.Y<416){
    if((int)Ball.X>Pad.X-8 && (int)Ball.X<Pad.X+64-8){
      // パドルのどこにボールが当たったのか？
      x=((int)(Ball.X+8-Pad.X)>>2)&0x0f;
      // 反発力テーブルを参照して，ボールの移動量を決定
      Ball.XAdd=PadDir[x][0]*2;
      Ball.YAdd=PadDir[x][1]*2;
    }
  }

  // ボールの移動&描画
  Ball.X+=Ball.XAdd;
  Ball.Y+=Ball.YAdd;
  rect.right=16;
  rect.bottom = 16;
  lpDDBack->BltFast((int)Ball.X, (int)Ball.Y, lpDDBall, &rect,
    DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);

  // ボールの移動範囲をチェック
  if(Ball.X<95){
    Ball.X=100;
    Ball.XAdd=-Ball.XAdd;
  }
  if(Ball.X>530){
    Ball.X=530;
    Ball.XAdd=-Ball.XAdd;
  }
  if(Ball.Y<0){
    Ball.Y=0;
    Ball.YAdd=-Ball.YAdd;
  }
  // ミスをしたか？
  if(Ball.Y>480-32)
    InitGame();

  // ゲームウェイト
  DWORD thisCount;
  do thisCount=GetTickCount();
  while(thisCount-lastCount<1000/120);

  // フリッピング
  lpDDPrimary->Flip(NULL,DDFLIP_WAIT);

  lastCount=thisCount;
}

// DirectDrawの初期化
BOOL InitDDraw(HWND hWnd)
{
  // DirectDrawオブジェクトの作成
  if(DirectDrawCreate(NULL,&lpDD,NULL)!=DD_OK) return FALSE;

  // 協調レベルの設定
  lpDD->SetCooperativeLevel(hWnd,
    DDSCL_EXCLUSIVE|DDSCL_FULLSCREEN);

  // 画面モードの設定
  lpDD->SetDisplayMode(640, 480, 16);

  // プライマリサーフェイスの初期化
  DDSURFACEDESC ddsd;
  DDSCAPS ddscaps;

  ZeroMemory(&ddsd, sizeof(ddsd));
  ZeroMemory(&ddscaps, sizeof(ddscaps));

  ddsd.dwSize=sizeof(ddsd);
  ddsd.dwFlags=DDSD_CAPS|DDSD_BACKBUFFERCOUNT;
  ddsd.ddsCaps.dwCaps=DDSCAPS_PRIMARYSURFACE|
    DDSCAPS_FLIP|DDSCAPS_COMPLEX;
  ddsd.dwBackBufferCount=1;

  lpDD->CreateSurface(&ddsd, &lpDDPrimary, NULL);

  ddscaps.dwCaps=DDSCAPS_BACKBUFFER;
  lpDDPrimary->GetAttachedSurface(&ddscaps, &lpDDBack);

  //  オフスクリーンサーフェイスの初期化
  ddsd.dwFlags=DDSD_CAPS|DDSD_WIDTH|DDSD_HEIGHT;
  ddsd.ddsCaps.dwCaps=DDSCAPS_OFFSCREENPLAIN;
  ddsd.dwWidth=64;
  ddsd.dwHeight=24;
  lpDD->CreateSurface(&ddsd, &lpDDPad, NULL);

  ddsd.dwWidth=16;
  ddsd.dwHeight=16;
  lpDD->CreateSurface(&ddsd, &lpDDBall, NULL);

  ddsd.dwWidth=640;
  ddsd.dwHeight=480;
  lpDD->CreateSurface(&ddsd, &lpDDBG, NULL);

  // プライマリとバックサーフェイスのクリア
  DDBLTFX ddbltfx;
  ZeroMemory(&ddbltfx, sizeof(DDBLTFX));
  ddbltfx.dwSize=sizeof(DDBLTFX);
  lpDDPrimary->Blt(NULL, NULL, NULL, DDBLT_COLORFILL|DDBLT_WAIT, &ddbltfx);
  lpDDBack->Blt(NULL, NULL, NULL, DDBLT_COLORFILL|DDBLT_WAIT, &ddbltfx);

  if(!ResumeScreen()){
    ReleaseObjects();
       MessageBox(hWnd,  "Surface  Restore  Error",  "Surface  Initialize",
MB_OK|MB_ICONHAND);
    return FALSE;
  }
  InitGame();

  return TRUE;
}
// ゲーム初期化
void InitGame()
{
  // ゲーム情報の初期化
  Pad.X=290;
  Ball.X=Ball.Y=50;
  Ball.XAdd=Ball.YAdd=2;
}
```

3次元で(ピン)ポン!

前編である今回の記事の大まかな流れとしては，DirectDrawの基本的な事項を簡単なサンプルゲーム(壁打ちピンポン)を例に挙げて説明する。そして，3Dを理解するうえで必要な周辺知識を解説してから，Direct3Dでベーシックなキューブを描画するプログラムを作成し，その説明をして，次号の後編で2Dのピンポンを3D化する準備としたい。読み進めるうちに自力で基本的な3Dアプリケーションを作れるレベルに達する……，というのが前後編共通の目的だ。まあ，とにかく勢いと気合で学ぶDirect3D講座なのだ。

なお，各章はほぼ独立した内容になっているので，面倒だと感じたらそ

この部分のプログラムは，基本的なスケルトンとしてとらえて，だいたいどのような動きをするのかがわかった時点で，飛ばし読みしてもらって構わない。あとで必要になったときに改めて読んで，少しずつ覚えていっても大丈夫だ(たいてい，初期化とかはどの本を読んでもプログラム的に変わるところはないし)。

Direct3DのためのDirectDraw予備的学習

まずDirectDrawを用いたサンプルゲームを見ていただきたい(画面1)。これは，シンプルな壁打ちピンポンで，左右のカーソルキーでパッドを操作し，ボールが落ちないようにひたすら打ち返すだけのゲームだ。ルールも単純ならプログラムもさらに単純で，コードの分量も異常といえるほど少ないので，サンプルとしては最適だと判断した。

さっそく，プログラムの解説に入りたいところだが，コンパイル法についてはコラム1を，Win32アプリケーションの仕組みについては，コラム2にまとめておいたので，各自必要に応じて参照しておいてほしい。

プログラムについて

アルゴリズム的にはそんなに難しいことはしていないが，注意してほしいのはパドルとボールの当たり判定のところだ。壁と同じように，パドルに当たったときに，単純にボール移動座標の入っている変数の符号を反転させるだけでは，45度の角度にしかボールは移動できない。そこでどうするかというと，あらかじめ飛ぶボールの方向を配列"PadDir"に用意しておく。そして，パドルの当たった場所に応じて，その配列を参照してボールの移動方向を決めてやればいい。こうすることによって，パドルの端に行けば行くほど，ボールの反射角が浅くなり，プレイヤーの操作によって，任意の場所にボールを飛ばしてやることが可能になる。

サーフェスとはなんぞ？

DirectDrawでは，表画面と裏画面の2つを取る。ディスプレイで見えるのが表画面で，実際にスプライトなどの画像を書き込むのは裏画面だ。つまり，裏画面にあらかじめ書いておいて，表画面にその内容をまとめて転送することで実際の表示が行われる。

一般的にダブルバッファリングといわれる手法で，ちらつきのない描画を実現する方法だ。前者をプライマリサーフェス，後者をバックサーフェスと呼ぶ。

スプライトなどのキャラクターデータを待避させて，バックサーフェスへ転送させる場所みたいなものがある。それはオフスクリーンサーフェスといい，基本的にメモリが許せばいくらでも取ることができる。スプライトなどのデータを大量に格納しておくバッファのようなものである。オフスクリーンサーフェスから，バックサーフェスへデータを転送して，スプライトの表示を行う。

もろもろの初期化

さて，実際にプログラムを見てみよう。InitDDraw関数内でいろいろと初期化をしている。

まずはDirectDrawの初期化関連だ。ここでは，DirectDrawオブジェクトの作成，協調レベルの設定，画面モードの設定といった一連の処理をしている。

DirectDrawはすべてのオブジェクトの大本だ。ここでCOMがウンタラ……といった小難しい話が出てくるのだが，自分で実際にCOMを作るのでなければ，そんなに意識することなく使えるので安心してほしい。要はDirectDrawというオブジェクトに，いろいろな関数がぶら下がっているというくらいの理解で大丈夫だ(関数のアドレスを集めたインタフェイス)。

DirectDrawCreate関数でDirectDrawオブジェクトを取得する。第1引数では，生成したいドライバのGUIDを指定する[*3]。ここでNULLを指定するとデフォルトのドライバを選択することになり，通常HALのドライバが選ばれる[*4]。第2引数には，確保しておいたIDirectDrawインタフェイスのポインタが入る。処理が成功するとDD_OKの戻り値を返す。

＊3　GUIDは，"Globally Unique Identifier"を略したもので，一意性を保証するように設計されたアルゴリズムで生成される128ビット(16バイト)の数値だ。乱暴ないい方だが，COMを識別するための値だと理解してほしい。

＊4　HAL(Hardware Abstraction Layer)はハードウェア支援で処理を高速に行う。逆にハードウェアの各機能をエミュレーションするのはHELであるが，死ぬほど遅いのでゲーム用としては実用的ではないだろう。

協調レベルの設定

SetCooperativeLevelで協調レベルの設定をする。第2引数のDDCSL_EXCLUSIVEで排他モードを，DDCSL_FULLSCREENでフル画面モードを指定している。排他モードとは，どの実行中のアプリケーションよりも優先度を高くするという意味だ。DDCSL_NORMALでウィンドウモードを選ぶことができるが，ここでは説明を省略する。

画面モードの設定とサーフェスの初期化

SetDisplayModeで，画面サイズは640×480ドット，16ビットカラー(65536色)を指定している。8ビットカラー(256色)の場合はさらにパレット設定が必要となってくるが，いまはほとんど16ビット以上のカラーしか使わないので，パレットの設定は省略してある。

次は先ほど説明をした各種サーフェスの取得である。まずはプライマリとバックのサーフェスを構築する。この辺もパラメータを含めて常套句ともいえる部分だ。簡単に説明するとDDSURFACEDESC構造体にそれぞれパラメータを入れてCreateSurfaceでサーフェスを設定する(DDSURFACEDESC構造体の詳細についてはオンラインヘルプなどを参照されたし)。ここでは，バックバッファを1枚指定してプライマリサーフェスを作り，バックバッファにアタッチしている。バックサーフェスはプライマリサーフェスの作成と同時に作られるので，特別にCreateSurfaceで作成する必要はない。

あとはオフスクリーンサーフェスだが，同じように必要なだけサーフェスを作成しておく。このサンプルでは，パッド，ボール，背景の3つのオフスクリーンサーフェスを用意している。

サーフェスの消失とその復旧

Alt+Ctrlキーなどでアプリケーションが切り替わったりすると，ビデオメモリが書き換えられてしまい，使用していたサーフェスの内容が破壊される場合がある。そこで元のアプリケーションに復帰したときには，サーフェスの修復をしなければいけない。Restoreで各サーフェスのメモリの再回復をし，再びビットマップの内容を読み込み，スプライトサーフェスへのコピーを行う。

サンプルでは，ウィンドウプロシージャ内のWM_ACTIVEAPPメッセージで呼び出されるResumeScreens関数でサーフェスの復旧をしている。と同時に，ビットマップの読み込みもしている。

スプライトの読み込みとカラーキーの設定

loadimageでビットマップをファイルからロードしている。GetDCでスプライトサーフェスのデバイスコンテキストを取得する。そして，CreateCompatibleDCでメモリ上にデバイスコンテキストを取得し，SelectObjectでリソースの内容を選択，すでに初期化したスプライト用サーフェスのDCを取得し，Bitbltで転送して解放する。

最後はカラーキーの設定だ。カラーキーとは，スプライトを描画するときに透明な色とされ，カラーキーで指定された部分は表示されない。これによって，重ね合わせ処理が意識することなく行える。ここでは無難に黒をカラーキーにしている。まあ，デバイスコンテキスト取得だの，オブジェクトの指定だの，初心者の方には難しいかもしれない。そういった方でも，リストを眺めていると，ここでファイル名が入ってビットマップを読み込んでいるなとか，流れ的にはだいたい理解できると思う。

Direct3 DRM

いろいろと長くなってしまったが，いよいよRMの解説に入ることにする。まずはDirect3Dの基本的な周辺知識の説明から始めよう。

コンピュータで3Dを表現するには，三角形や四角形などの多角形(ポリゴン)を1つひとつ組みあわせて，物体を表現するのが一般的だ。各ポリゴンの頂点を結んで面(フェイス)を形成し[*5]，その面で構成された図形をメッシュという。

そして3Dというからには，縦横を示すXY座標に加え，奥行きを示すZ座標の3つの座標軸を使用する。数学で習った座標系とはZ軸が逆になっていることに注意してほしい。Z軸が視点から離れる方向へと，値が大きくなっているのだ。Direct3Dではこの「左手座標系」を採用している。この座標系のひとつの単位を「フレーム」という概念で管理する。特に，画面の基準となる絶対座標を「シーンフレーム」(または単にシーン)と呼ぶ。このシーンフレームを基準に，各オブジェクト(メッシュ)をフレームに載せて，フレームを配置して3次元世界を構築していく。

ちなみにこのフレームは，入れ子にすることができ，親フレームを動かすとそれに付随する子フレームも動くことになる。具体的には，人間の関節を表現したいときに使うと，とても便利な方法だ。あと書き忘れていたが，法線というものがある。これは面の向きを指定するベクトルだ。

3次元空間をのぞき込む窓のようなものが「ビューポート」で，シーンがレンダリングされる場所と方法を定義する。つまり，ビューポートの位置とビューポートになにを記述するのかを決める。そして実際に画像が表示される「デバイス」[*6]に取りつけて，レンダリングをする。デバイスには，品質などのさまざまなレンダリングオプションを作成する。

＊5　フェースは凹んだ部分ができない限り，頂点をいくつでも持つことができる。
＊6　デバイスは本質的には，バックバッファのオブジェクトそのものだ。つまり，このインタフェイスが表すのは画面そのものなのだ。

ライトについて

次にこのメッシュを照らす各種ライトを見てみよう。このライトがないとせっかく作ったモノもお先真っ暗である。

環境ライト(ambient light)

これは一般的にアンビエントライトともいわれる。シーンのすべてのオブジェクトに均等に光が届き，同じ明るさで空間を満たす。

平行ライト(directional light)

特定の方向から一様に注いでくる光。これは「地上における太陽の光のようなもの」と考えると理解しやすいだろう。

スポットライト(spot light)

これは読んで字のごとく，一点から円錐状に広がる光だ。

点ライト(point light)

1点から全方向へ放たれる光。暗闇の中で，光を発する白熱電球を想像するといいだろう。

平行点ライト(parallel light)

メッシュ単位で平行ライトになる点ライト。性質上，平行ライトと点ライトに似ているが，処理速度が速いということで異なる。点ライトは，頂点ごとに計算をしているので，ポリゴンが増えるとどうしても処理が重くなってしまうのだ。これらのライトはフレームとしてくっつけて，各フレームを照らすことになる。

シェーディング

物体に陰影をつけて，3次元的な立体感を出す着色を行う処理。フラットシェーディングは，面全体のポリゴンを同じ明るさで，塗りつぶす。いちばん単純で高速な方法だが，ややカクカクした感じになる。グローシェーディングは，平面のそれぞれの頂点について演算を行い，平均化して面の明るさがなだらかに変化する。わりと高速なのに，そこそこのクオリティが出るので，現在もっとも一般的に使われている手法だ。

最後にもっともクオリティが高いのが，フォンシェーディングだ。ポリゴン上の点すべてに対して明るさを求める。もちろん，3つのうちでもっとも処理が重い。Direct3Dではそのモードは予約されているもののまだサポートされていない(編注：もっと重くて表現力の優れたPerPixelShaderはサポートされているのだが)。

テクスチャ

Direct3Dにおいてメッシュに直接ビットマップを拡大/縮小したりして張りつけることを「テクスチャマッピング」という。ポリゴンだけで目的のオブジェクトを生成するのは，かなり処理的に無理がある。たとえば地球を表現したいとする，そこで世界地図のビットマップを用意しておいて，球状のメッシュに貼りつけると，あっさりと地球ができあがるというわけだ。このように，テクスチャを有効に利用することで，あまり処理を重くすることなくポリゴンの表現力を格段に上げることができる。ただしテクスチャのサイズは，2の累乗(64とか)に限られているので注意。あとテクスチャの座標系は，縦横の座標をそれぞれ"u,v"で表し，左上(0，0)から右下(1，1)までの値を取る。

3Dサンプルの解説

さておおまかに3Dとはどういうものなのかを概説してきたが，ここからは，実際にポリゴンを表示するプログラムを解説しながら，RMの使用法を学んでいこう。

このサンプルはキューブ(立方体)を描画して，カーソルキーでぐりぐり動かすだけの，基本中の基本ともいうべき3Dプログラムである。

初期化

まずDirectDrawを初期化するのだが，いくつか重要な変更点がある。前のDirectDrawサンプルのように2Dの機能だけを使うのなら，

```
DirectDrawCreate(NULL,&lpDD,NULL);
```

のように現在アクティブなDirectDrawデバイスを使うだけで，ほとんど問題はなかった。しかし，たとえば普段使っている(プライマリ)ビデオカードは3Dの機能が低く，より3D機能が高い別の(セカンダリ)ビデオカードも入れている場合，これでは機能が低いほうのデバイスが選ばれてしまうのだ。

そこでDirect3Dを使うときは，ハードウェアで3Dをより機能を高度にサポートしているデバイスを調べて選択したほうが確実だ(最近のビデオカードは2Dも3Dも高機能なものが多いのだが)。

DirectDrawEnumerate関数で，インストールされているビデオカードのDirectDrawドライバの列挙を行う。第1引数で指定したコールバック関数(このサンプルでは，DDEnumCallback関数)を呼び出し，次々とDirectDrawのGUIDを列挙する。これで，より3Dに適したDirectDrawデバイスを選定できる。

あと，プライマリサーフェス作成時のCapsにDDSCAPS_3DDEVICEを加えている。これは「3Dを使いますよ」ということをOSに伝えているところだ。

次にDirect3Dの初期化である(InitD3D関数)。Direct3Dオブジェクトを作成する。DirectDrawで最初にDirectDrawオブジェクトを作成するのと同様に，Direct3Dのオブジェクトを作らなければならない。Direct3Dオブジェクトを取得するために，DirectDrawのCOMからこの関数を呼び出している[*7]。

つまり，Direct3DもDirectDrawのなかに存在しているということだ。QueryInterfaceでDirect3Dオブジェクトの取得を行う。続いてここでも，

列挙によってDirect3Dデバイスを取得している。あとは，Zバッファを作成して，バックバッファへアタッチしている。

続いてDirect3DRMの初期化である。ここでも，QueryInterfaceでRMオブジェクトを取得し，RMデバイス，シーンフレーム，カメラフレーム，ビューポートの作成，バッククリップの設定を行っている。バッククリップとは，奥行きをどこまで表示するかというもので，ここでは適当に奥行き500.0まで表示するようにしている。あと，説明が遅れたが"D3DVAL"とは，Direct3D用の値(Value)という意味だ。

オブジェクトの配置

ここからいよいよオブジェクト配置についての説明に入る(BuildScene関数)。まずはクオリティの設定から，影の陰影を32段階，テクスチャの発色数を256，テクスチャの陰影を32段階，ディザ(境界色を滑らかにする)を有効に，テクスチャにはバイリニアフィルタ(拡大時のドット補間)をかけるようにする。ライトは，環境ライトと点ライトを作成したライトフレームに取り付ける(AddLightメソッド)。

＊7　COMを扱ううえで特に気をつけなければならないのは，不要になったオブジェクトは解放しなくてはならないということだ。このプログラムでは，REREASEマクロを使い，各オブジェクトを解放している。

キューブモデルの作成

メッシュを作成するには，CreateMeshBuilderを使う。

リスト2

```
#include "stdafx.h"
#include "Cube.h"
#define APPNAME "Cube"

LRESULT CALLBACK WndProc(HWND, UINT, UINT, LONG);
BOOL bActive = FALSE;

IDirectDraw *lpDD=NULL;                    // DirectDrawオブジェクト
IDirectDrawSurface *lpDDPrimary=NULL;      // プライマリサーフェス
IDirectDrawSurface *lpDDBack=NULL;         // バックサーフェス

IDirect3D2 *lpD3D=NULL;                    // Direct3Dオブジェクト
IDirect3DDevice2 *lpD3DDev=NULL;           // Direct3Dデバイス
IDirectDrawSurface *lpDDZ=NULL;            // Zバッファ

IDirect3DRM3 *lpD3DRM=NULL;                // Direct3D RMオブジェクト
IDirect3DRMDevice3 *lpD3DRMDev=NULL;       // Direct3D RMデバイス

IDirect3DRMFrame3 *lpD3DRMScene=NULL;  // シーンフレーム
IDirect3DRMFrame3 *lpD3DRMCamera=NULL; // カメラフレーム
IDirect3DRMViewport2 *lpD3DRMView=NULL;// ビューポート
IDirect3DRMFrame3 *lpD3DRMLight=NULL;  // ライトフレーム

IDirect3DRMFrame3 *lpD3DRMCube=NULL;   // キューブオブジェクト

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmdLine, int nCmdShow)
{
  WNDCLASS wc;
  HWND hWnd;
  MSG msg;

  wc.style=CS_HREDRAW|CS_VREDRAW;
  wc.lpfnWndProc=WndProc;
  wc.cbClsExtra=0;
  wc.cbWndExtra=0;
  wc.hInstance=hInstance;
  wc.hIcon=LoadIcon(hInstance, IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground=(HBRUSH)GetStockBrush(BLACK_BRUSH);
  wc.lpszMenuName=0;
  wc.lpszClassName=APPNAME;

  RegisterClass(&wc);

  // ウィンドウクラスの登録
  hWnd=CreateWindowEx(0, APPNAME, APPNAME, WS_POPUP, 0, 0,
    GetSystemMetrics(SM_CXSCREEN), GetSystemMetrics(SM_CYSCREEN),
    NULL, NULL, hInstance, NULL);

  if(!hWnd) return FALSE;

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);

  // DirectDrawの初期化
  if(!InitDDraw(hWnd)) return FALSE;
  // Direct3Dの初期化
  if(!InitD3D(hWnd)) return FALSE;
  // シーンの初期化
  if(!BuildScene(lpD3DRM, lpD3DRMDev, lpD3DRMView,
lpD3DRMScene, lpD3DRMCamera)) return FALSE;

  // 初期化終了
  bActive=TRUE;

  Render();
  BOOL update;
  // メッセージポンプ
  while(TRUE){
    if(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE)){
      if(!GetMessage(&msg, NULL, 0, 0 )){
        return msg.wParam;
      }
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
    update=FALSE;
    if(bActive){
      if(GetAsyncKeyState(VK_UP)){
         lpD3DRMCube->AddRotation(D3DRMCOMBINE_AFTER, D3DVAL(1), D3DVAL(0),
D3DVAL(0), D3DVAL(0.1));
        update=TRUE;
      }
      if(GetAsyncKeyState(VK_DOWN)){
         lpD3DRMCube->AddRotation(D3DRMCOMBINE_AFTER, D3DVAL(1), D3DVAL(0),
D3DVAL(0), -D3DVAL(0.1));
        update=TRUE;
      }
      if(GetAsyncKeyState(VK_LEFT)){
         lpD3DRMCube->AddRotation(D3DRMCOMBINE_AFTER, D3DVAL(0), D3DVAL(1),
D3DVAL(0), D3DVAL(0.1));
        update=TRUE;
      }
      if(GetAsyncKeyState(VK_RIGHT)){
         lpD3DRMCube->AddRotation(D3DRMCOMBINE_AFTER, D3DVAL(0), D3DVAL(1),
D3DVAL(0), -D3DVAL(0.1));
        update=TRUE;
      }
      if(update) Render();
    }
  }
}

long FAR PASCAL WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  switch(message){
  case WM_ACTIVATEAPP:
    bActive=(BOOL)wParam;
    if(bActive && lpDD){
      ResumeSurfaces();
      Render();
    }
    break;
  case WM_CREATE:
    break;
  case WM_SETCURSOR:
    SetCursor(NULL);
    return TRUE;
  case WM_KEYDOWN:
    switch(wParam){
    case VK_ESCAPE:
      DestroyWindow(hWnd);
      break;
    }
    return TRUE;
  case WM_DESTROY:
    bActive=FALSE;
    ReleaseObjects();
    PostQuitMessage(0);
    break;
  }
  return DefWindowProc(hWnd, message, wParam, lParam);
}

// オブジェクトの開放
void ReleaseObjects()
{
  RELEASE(lpD3DRMCube);
  RELEASE(lpD3DRMLight);
  RELEASE(lpD3DRMView);
  RELEASE(lpD3DRMCamera);
  RELEASE(lpD3DRMScene);
  RELEASE(lpD3DRMDev);
  RELEASE(lpD3DRM);
  RELEASE(lpD3DDev);
```

```
  RELEASE(lpDDZ);
  RELEASE(lpD3D);
  RELEASE(lpDDBack);
  RELEASE(lpDDPrimary);
  RELEASE(lpDD);
}

// DirectDrawドライバの列挙と選定
BOOL FAR PASCAL DDEnumCallback(GUID FAR *lpGUID, LPSTR lpDriverDesc,
  LPSTR lpDriverName, LPVOID lpContext)
{
  LPDIRECTDRAW lpDD;
  DDCAPS DriverCaps, HELCaps;

  if(lpGUID){
    // DirectDrawオブジェクトを試しに作成
    if(DirectDrawCreate(lpGUID, &lpDD, NULL)!=DD_OK)
return DDENUMRET_OK;

    memset(&DriverCaps,0,sizeof(DDCAPS));
    DriverCaps.dwSize=sizeof(DDCAPS);
    memset(&HELCaps, 0, sizeof(DDCAPS));
    HELCaps.dwSize=sizeof(DDCAPS);

    // 能力を取得
    if(lpDD->GetCaps(&DriverCaps, &HELCaps)!=DD_OK){
      lpDD->Release();
      return DDENUMRET_OK;
    }

    // 3Dアクセラレーションが利用できる場合で,
    // さらにテクスチャが使える場合は, それを採用
    if((DriverCaps.dwCaps&DDCAPS_3D)
        &&(DriverCaps.ddsCaps.dwCaps&DDSCAPS_TEXTURE)){
        *(LPDIRECTDRAW*)lpContext = lpDD;
        return DDENUMRET_CANCEL;
    }
    lpDD->Release();
  }
  // 他のドライバを試す
  return DDENUMRET_OK;
}

// DirectDrawの初期化
BOOL InitDDraw(HWND hWnd)
{
  // DirectDrawドライバを列挙
  if(DirectDrawEnumerate(DDEnumCallback,&lpDD)!=DD_OK) return FALSE;

  // HALが利用できない場合, 現在のアクティブなドライバを使う
  if(!lpDD)
    if(DirectDrawCreate(NULL, &lpDD, NULL)!=DD_OK) return FALSE;

  // 協調レベルの設定
  lpDD->SetCooperativeLevel(hWnd,DDSCL_EXCLUSIVE|DDSCL_FULLSCREEN);

  // 画面モードの設定
  lpDD->SetDisplayMode(640, 480, 16);

  // サーフェイスの作成
  DDSURFACEDESC ddsd;
  DDSCAPS ddscaps;

  ZeroMemory(&ddsd, sizeof(ddsd));
  ZeroMemory(&ddscaps, sizeof(ddscaps));

  ddsd.dwSize=sizeof(ddsd);
  ddsd.dwFlags=DDSD_CAPS|DDSD_BACKBUFFERCOUNT;
  ddsd.ddsCaps.dwCaps=DDSCAPS_PRIMARYSURFACE|DDSCAPS_3DDEVICE|
    DDSCAPS_FLIP|DDSCAPS_COMPLEX;
  ddsd.dwBackBufferCount=1;

  lpDD->CreateSurface(&ddsd,&lpDDPrimary,NULL);

  ddscaps.dwCaps=DDSCAPS_BACKBUFFER;
  lpDDPrimary->GetAttachedSurface(&ddscaps,&lpDDBack);

  DDBLTFX ddbltfx;
  ZeroMemory(&ddbltfx, sizeof(DDBLTFX));
  ddbltfx.dwSize=sizeof(DDBLTFX);
  lpDDPrimary->Blt
(NULL, NULL, NULL, DDBLT_COLORFILL|DDBLT_WAIT, &ddbltfx);
  lpDDBack->Blt
(NULL, NULL, NULL, DDBLT_COLORFILL|DDBLT_WAIT, &ddbltfx);

  return TRUE;
}

// DirectDraw初期化エラー処理
BOOL InitDDError(HWND hWnd)
{
  ReleaseObjects();
  MessageBox
(hWnd, "Initialize Error", "DirectDraw Initialize", MB_OK|MB_ICONHAND);
  DestroyWindow(hWnd);
  return FALSE;
}

// Direct3Dデバイスを列挙する
HRESULT FAR PASCAL D3DEnumCallback(LPGUID lpGuid, LPSTR lpDeviceDescription,
  LPSTR lpDeviceName, LPD3DDEVICEDESC lpHWDesc, LPD3DDEVICEDESC lpHELDesc,
  LPVOID lpContext)
{
  LPGUID *lpguid=(LPGUID*)lpContext;

  // ハードウェアデバイスか?
  if(lpHWDesc->dcmColorModel){
    // ハードウェアならGUIDを保存して採用
    *lpguid=lpGuid;
    return D3DENUMRET_CANCEL;
  }
  return D3DENUMRET_OK;
}

// Direct3Dの初期化
BOOL InitD3D( HWND hWnd )
{
  BOOL bHal;
  LPGUID lpguid=NULL;
  DDSURFACEDESC ddsd;

  // Direct3Dオブジェクトの取得
   if(lpDD->QueryInterface(IID_IDirect3D2, (void**)&lpD3D )!=DD_OK) return
InitD3DError(hWnd);

  // デバイスのGUIDの取得
  lpD3D->EnumDevices(D3DEnumCallback, &lpguid);

  bHal=lpguid!=NULL;

  // Zバッファの作成
  memset(&ddsd, 0, sizeof(DDSURFACEDESC));
  ddsd.dwSize=sizeof(DDSURFACEDESC);
  ddsd.dwFlags=DDSD_WIDTH|DDSD_HEIGHT|DDSD_CAPS|DDSD_ZBUFFERBITDEPTH;
  ddsd.dwWidth=640;
  ddsd.dwHeight=480;
  ddsd.dwZBufferBitDepth=16;
  ddsd.ddsCaps.dwCaps=DDSCAPS_ZBUFFER;
  ddsd.ddsCaps.dwCaps|=(bHal?DDSCAPS_VIDEOMEMORY:DDSCAPS_SYSTEMMEMORY);
  lpDD->CreateSurface(&ddsd, &lpDDZ, NULL);

  // Zバッファをバックバッファにアタッチ
  lpDDBack->AddAttachedSurface(lpDDZ);

  if(bHal){
    // ハードウェアデバイスの作成
    if(lpD3D->CreateDevice(*lpguid, lpDDBack, &lpD3DDev)!=D3D_OK)
      return InitD3DError(hWnd);
  } else {
    // もしもHALではなかったらをMMXデバイスの作成
                  if(lpD3D->CreateDevice(IID_IDirect3DMMXDevice,    lpDDBack,
&lpD3DDev)!=D3D_OK)
      return InitD3DError(hWnd);
  }

  // Direct3D RMオブジェクトの生成
  IDirect3DRM *lpOldD3DRM;
  if(Direct3DRMCreate(&lpOldD3DRM)!=D3D_OK) return InitD3DError(hWnd);
      HRESULT   result=lpOldD3DRM->QueryInterface(IID_IDirect3DRM3,   (void
**)&lpD3DRM);
  RELEASE(lpOldD3DRM);
  if(result!=D3D_OK) return InitD3DError(hWnd);

  // Direct3D RMデバイスの作成
  if(lpD3DRM->CreateDeviceFromD3D(lpD3D, lpD3DDev, &lpD3DRMDev)!=D3D_OK)
    return InitD3DError(hWnd);

  // シーンフレームの作成
  lpD3DRM->CreateFrame(NULL, &lpD3DRMScene);
  lpD3DRMScene->SetPosition(NULL, D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0));

  // カメラフレームの作成
  lpD3DRM->CreateFrame(lpD3DRMScene, &lpD3DRMCamera);
  lpD3DRMCamera->SetPosition(lpD3DRMScene, D3DVAL(0.0), D3DVAL(0.0), D3DVAL(-
10.0));

  // ビューポートの作成
  if(lpD3DRM->CreateViewport(lpD3DRMDev, lpD3DRMCamera, 0, 0,
    lpD3DRMDev->GetWidth(), lpD3DRMDev->GetHeight(), &lpD3DRMView)!=DD_OK)
    return InitD3DError( hWnd );

  // バッククリップの設定
  if(lpD3DRMView->SetBack(D3DVAL(500.0))!=DD_OK )
 return InitD3DError(hWnd);

  return TRUE;
}

// Direct3D初期化エラー処理
BOOL InitD3DError(HWND hWnd)
{
  ReleaseObjects();
```

```
  MessageBox
(hWnd, "Initialize Error", "Direct3D Initialize", MB_OK|MB_ICONHAND);
  DestroyWindow(hWnd);
  return FALSE;
}

// サーフェスの修復
BOOL ResumeSurfaces()
{
  lpDDPrimary->Restore();
  lpDDZ->Restore();
  return TRUE;
}

// 頂点のリスト
D3DVECTOR vertices[]={
  D3DVAL(-0.5), D3DVAL(-0.5), D3DVAL(-0.5),
  D3DVAL(-0.5), D3DVAL(-0.5), D3DVAL(0.5),
  D3DVAL(-0.5), D3DVAL(0.5), D3DVAL(-0.5),
  D3DVAL(-0.5), D3DVAL(0.5), D3DVAL(0.5),
  D3DVAL(0.5), D3DVAL(-0.5), D3DVAL(-0.5),
  D3DVAL(0.5), D3DVAL(-0.5), D3DVAL(0.5),
  D3DVAL(0.5), D3DVAL(0.5), D3DVAL(-0.5),
  D3DVAL(0.5), D3DVAL(0.5), D3DVAL(0.5)
};

// 法線ベクトルのリスト
D3DVECTOR normals[]={
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1.0),
  D3DVAL(0.0), D3DVAL(1.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(1.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(1.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(1.0), D3DVAL(0.0),
  D3DVAL(1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(-1.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(-1.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(-1.0),
  D3DVAL(0.0), D3DVAL(0.0), D3DVAL(-1.0),
  D3DVAL(0.0), D3DVAL(-1.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(-1.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(-1.0), D3DVAL(0.0),
  D3DVAL(0.0), D3DVAL(-1.0), D3DVAL(0.0),
  D3DVAL(-1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(-1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(-1.0), D3DVAL(0.0), D3DVAL(0.0),
  D3DVAL(-1.0), D3DVAL(0.0), D3DVAL(0.0)
};

// フェースリスト
DWORD face[]={
  4, 1, 0, 5, 1, 7, 2, 3, 3,
  4, 2, 4, 3, 5, 7, 6, 6, 7,
  4, 5, 8, 4, 9, 6, 10, 7, 11,
  4, 0, 12, 2, 13, 6, 14, 4, 15,
  4, 0, 16, 4, 17, 5, 18, 1, 19,
  4, 0, 20, 1, 21, 3, 22, 2, 23,
0};

BOOL BuildScene(IDirect3DRM3 *lpD3DRM, IDirect3DRMDevice3 *lpD3DRMDev,
  IDirect3DRMViewport2 *lpD3DRMView, IDirect3DRMFrame3 *lpD3DRMScene,
  IDirect3DRMFrame3 *lpD3DRMCamera)
{

  // クオリティの設定
  lpD3DRMDev->SetShades(32);
  lpD3DRM->SetDefaultTextureColors(256);
  lpD3DRM->SetDefaultTextureShades(32);
  lpD3DRMDev->SetDither(TRUE);
  lpD3DRMDev->SetTextureQuality(D3DRMTEXTURE_LINEAR);

  // 光源のフレーム作成
  lpD3DRM->CreateFrame(lpD3DRMScene, &lpD3DRMLight);
  lpD3DRMLight->SetPosition(lpD3DRMScene, D3DVAL(1.0), D3DVAL(1.0),D3DVAL(-
3.0));

  // 光源の作成
  LPDIRECT3DRMLIGHT pLight;

  // 点ライト
  lpD3DRM->CreateLightRGB(D3DRMLIGHT_POINT, D3DVAL(0.8), D3DVAL(0.8),
    D3DVAL(0.8), &pLight);
  lpD3DRMLight->AddLight(pLight);
  RELEASE(pLight);

  // 環境ライト
  lpD3DRM->CreateLightRGB(D3DRMLIGHT_AMBIENT, D3DVAL(0.2),
    D3DVAL(0.2), D3DVAL(0.2), &pLight);
  lpD3DRMLight->AddLight(pLight);
  RELEASE(pLight);

  // キューブの作成
  lpD3DRM->CreateFrame(lpD3DRMScene, &lpD3DRMCube);
  lpD3DRMCube->SetPosition(lpD3DRMScene, D3DVAL(0.0), D3DVAL(0.0),
    D3DVAL(0.0));
    lpD3DRMCube->SetOrientation(lpD3DRMScene,  D3DVAL(0.0),  D3DVAL(-0.1),
D3DVAL(0.3),
    D3DVAL(0), D3DVAL(0.3), D3DVAL(0.1));
  LPDIRECT3DRMMESHBUILDER3 builder;
  LPDIRECT3DRMTEXTURE3 texture;

  // テクスチャの読み込み
  lpD3DRM->LoadTexture("TEXTURE.BMP", &texture);

  // メッシュの作成
  lpD3DRM->CreateMeshBuilder(&builder);
  builder->AddFaces(8, vertices, 6*4, normals, face, NULL);
  builder->SetColor(D3DRGBA(1.0,1.0,1.0,0.8));
  builder->SetQuality(D3DRMRENDER_GOURAUD);
  builder->Scale(D3DVAL(2.0), D3DVAL(2.0), D3DVAL(2.0));

  // テクスチャ座標の設定
  builder->SetTextureCoordinates(0, D3DVAL(0), D3DVAL(1));
  builder->SetTextureCoordinates(1, D3DVAL(1), D3DVAL(1));
  builder->SetTextureCoordinates(2, D3DVAL(0), D3DVAL(0));
  builder->SetTextureCoordinates(3, D3DVAL(1), D3DVAL(0));
  builder->SetTextureCoordinates(4, D3DVAL(1), D3DVAL(1));
  builder->SetTextureCoordinates(5, D3DVAL(0), D3DVAL(1));
  builder->SetTextureCoordinates(6, D3DVAL(1), D3DVAL(0));
  builder->SetTextureCoordinates(7, D3DVAL(0), D3DVAL(0));
  builder->SetTexture(texture);
  builder->SetPerspective(TRUE);

  lpD3DRMCube->AddVisual(builder);

  RELEASE(texture);
  RELEASE(builder);

  return TRUE;
}

void Render()
{
  // ビューのクリア
  lpD3DRMView->Clear(D3DRMCLEAR_ALL);
  // レンダリング
  lpD3DRMView->Render(lpD3DRMScene);
  // レンダリングした結果をバックバッファへコピー
  lpD3DRMDev->Update();
  // フリッピング
  lpDDPrimary->Flip(NULL, DDFLIP_WAIT);
}
```

あらかじめ用意しておいた配列データfaceにAddFacesメソッドでMeshBuilderに追加し，SetColorRGBで白色に，そしてScaleで2倍に拡大している。テクスチャはLoadTexureでファイルから読み込み，SetTextureCoordinatesでテクスチャとフェイスを関連付け，SetPerspectiveでパースペクティブコレクト（テクスチャの歪みを補正する機能）をかけている。そしてSetTextureでテクスチャをセットだ。

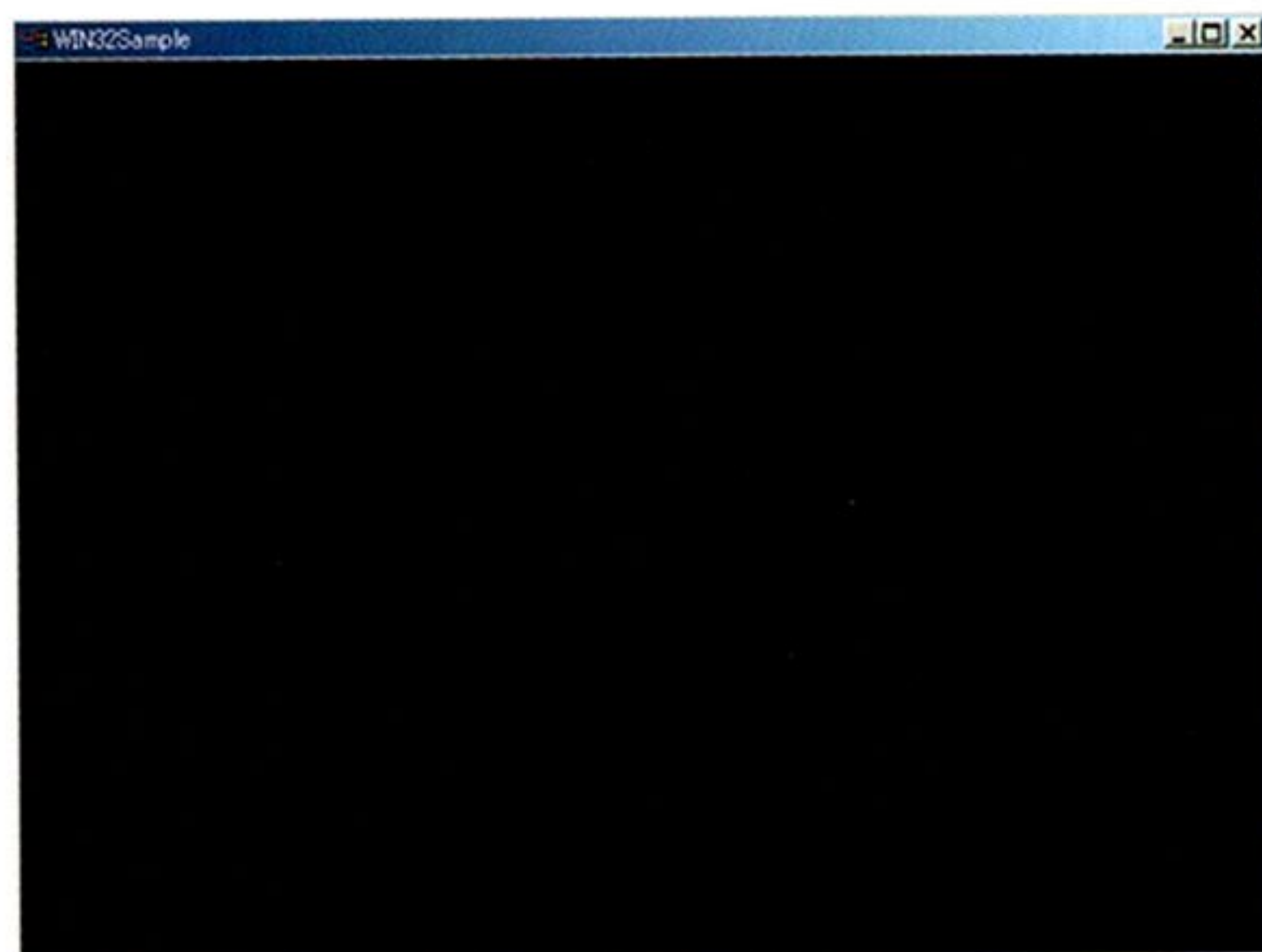
あとはメッシュの表示だ（Render関数）。ここは流れ作業的にClearでビューのクリアをしておき，レンダリングしてプライマリとバックバッファをフリップする。

これで，カーソルキーが押されるとAddRotation関数でキューブが回転する。

終わりに

以上，駆け足でDirectDrawとDirect3DRMについて説明したが，どうだっただろうか。この説明を読んで，自分でサンプルプログラムをいじったりして，流れ的にだいたいは理解できると思う。誌面の都合上，いろいろとはしょってしまった感もなきにしもあらずなので，疑問を持たれたかもしれないが，その疑問点を出発点にして，「なにがわからないかわからない」状態から抜け出すことはできるように書いたつもりだ。あとはそれがきっかけとなり，自分で簡単に調べられるはず（だよね）。

次回は，いよいよピンポンを3D化してみる予定だ。では，また次号でお会いしよう。

画面2 3D表示プログラムの基本中の基本がこれ。テクスチャを貼ったポリゴンオブジェクトを画面に表示して，それを動かせればあとは応用次第でなんでもできる

WIN32Sample

画面3 単にウィンドウを開くだけのサンプルなので，あまり期待して実行しないように

リスト3

```
// WIN32アプリケーションサンプル
// WIN32Sample.cpp

#include "stdafx.h"
#define APPNAME "WIN32Sample"

HINSTANCE hAppInst;

LRESULT CALLBACK  WndProc(HWND, UINT, UINT, LONG);

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{

  WNDCLASS wc;
  HWND hWnd;
  MSG msg;

  wc.style=CS_HREDRAW|CS_VREDRAW;
  wc.lpfnWndProc=WndProc;
  wc.cbClsExtra=0;
  wc.cbWndExtra=0;
  wc.hInstance=hInstance;
  wc.hIcon=NULL;
  wc.hCursor=LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground=(HBRUSH)GetStockObject(BLACK_BRUSH);
  wc.lpszMenuName=0;
  wc.lpszClassName=APPNAME;

  RegisterClass(&wc);

  hAppInst=hInstance;
  // ウィンドウクラスの登録
  hWnd=CreateWindowEx(0,APPNAME,APPNAME,WS_OVERLAPPEDWINDOW,
      0,0,640,480,NULL,NULL,hInstance,NULL);

  if(!hWnd) return FALSE;

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);

  // メッセージポンプ
  while(GetMessage(&msg, NULL, 0, 0)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
}

long FAR PASCAL WndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{
  switch(message){

  case WM_KEYDOWN:
    switch(wParam){

  case VK_ESCAPE:
      DestroyWindow(hWnd);
      break;
    }
    return TRUE;

  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  }
  return DefWindowProc(hWnd, message, wParam, lParam);
}
```

COLUMN 2

Win32アプリケーションについて

DirectXを使ったゲーム開発では，MFCを使用せず自前でウィンドウポンプやメッセージを作るのが一般的だ。つまり，C++(ほぼCの機能)とDirectX SDKだけで開発するわけだ(別にMFCを使ってもいいのだが)。

Windowsアプリケーションの処理の流れは，DOS上でのプログラムがMain関数で始まるのと同じで，Win32アプリケーションでは，まずはWinMain関数が最初に呼ばれる。この関数内でウィンドウの初期化・登録を行い，ウィンドウプロシージャ(ウィンドウ関数)の設定をする。そしてここで受け取ったメッセージ(画面の描画やキーボードが押されているかなどの指示)をウィンドウプロシージャ(Windowsから受け取ったメッセージを振り分けるコールバック関数)で実際の処理を実行し，メッセージポンプへ戻り，再びWindowsからのメッセージを待つ。

……まあ，字面だけではわかりにくいと思うので，具体的にプログラムを挙げて説明しよう(リスト3，画面3)。

このサンプルは，単純にウィンドウを画面に表示させているだけのWin32アプリケーションの定型といえるプログラムだ。

では簡単に順を追って説明していこう。最初の#defineでこのアプリケーションの名前を定義しておく。そして次に件のWinMain関数の中身を見ていこう。WNDCLASS構造体でウィンドウの初期化をしている。ここでは主に，ウィンドウプロシージャの設定のところさえ押さえておけば大丈夫だ(wc.lpfnWndProc=WndProc;の個所)。ほかのところは，必要に応じてオンラインヘルプなどのリファレンスを参考にしてほしいが，まあデフォルトのままでなんとかなるだろう。

CreateWindowEx関数でウィンドウの設定をして(サイズなど。ここでは640×480ドットとしている)，次のRegisterClass関数でウィンドウクラスの登録だ。"if(!hWnd) return FALSE;"でエラーチェックをしたあと，"ShowWindow(hWnd, nCmdShow);"によってウィンドウの表示状態を設定し，"UpdateWindow(hWnd);"でウィンドウの領域を更新する。

次にいよいよいちばん重要なメッセージポンプが出てくる。ここでは，MSG構造体のmsgのなかにメッセージの内容が入っているので，GetMessage関数で，そのメッセージを受け取り，ループ内のTranslateMessage関数でメッセージを解釈し，次のDispatch Message関数ではその解釈したメッセージとともに，ウィンドウプロシージャ(WndProc)へ渡す。

ウィンドウプロシージャWndProcは，実際の処理をSwitch文で振り分けているコールバック関数だ。ここでは，もしキーボードが押されていた場合，"WM_KEYDOWN"(このWM_～という部分はメッセージのキーワードのようなもので，オンラインヘルプなんかで検索をかけるとメッセージがたくさん出てくる)メッセージで，エスケープキーが押されていると判断するとウィンドウを破棄して終了処理に入る。

……とまあ大雑把な説明であったが，サンプルリストを読み，実行をトレースすることによってWindowsアプリケーションの大まかな仕組みのようなものが理解できると思う。まあ，ウィンドウプロシージャの部分がいちばん重要で，ここさえ書き換えればほかのところはデフォルトのままでも，自力でプログラムを組めるようになるだろう。この講座でも，ここで作成したものをスケルトンとしてほかのサンプルプログラムでも流用している。

特別企画　オリジナルマシンを作ろう 第3弾

そしてオリジナルデザインへ X Thunderbird登場

菊地 功　Kikuchi Isawo

「オリジナルPCケースを作ろう」シリーズも第3弾。前回のアルミ板というPCケースに最適な素材と板金曲げ器導入により，この企画も実用段階に到達したといっていいだろう。問題はデザインである。いまだマンハッタンシェイプ以上のCOOLなデザインに出合っていない。
X68000のシャープなイメージを残しつつ，拡張性に富み，多少でもオリジナリティのある形状は作れないか？　より力強くあるべきマンハッタンシェイプの進化形として，今回はAthlonとノーマルATXマザーをベースに，搭載デバイスでも妥協のない構成で，拡張トリプルタワー形状の実現を試みた。
トリッキーに配置されたタワー構成により，拡張スロットフル増設も可能（ポップアップハンドルは絶対あったほうがいいな……）。ポストマンハッタンへ向けてのひとつの提言である。

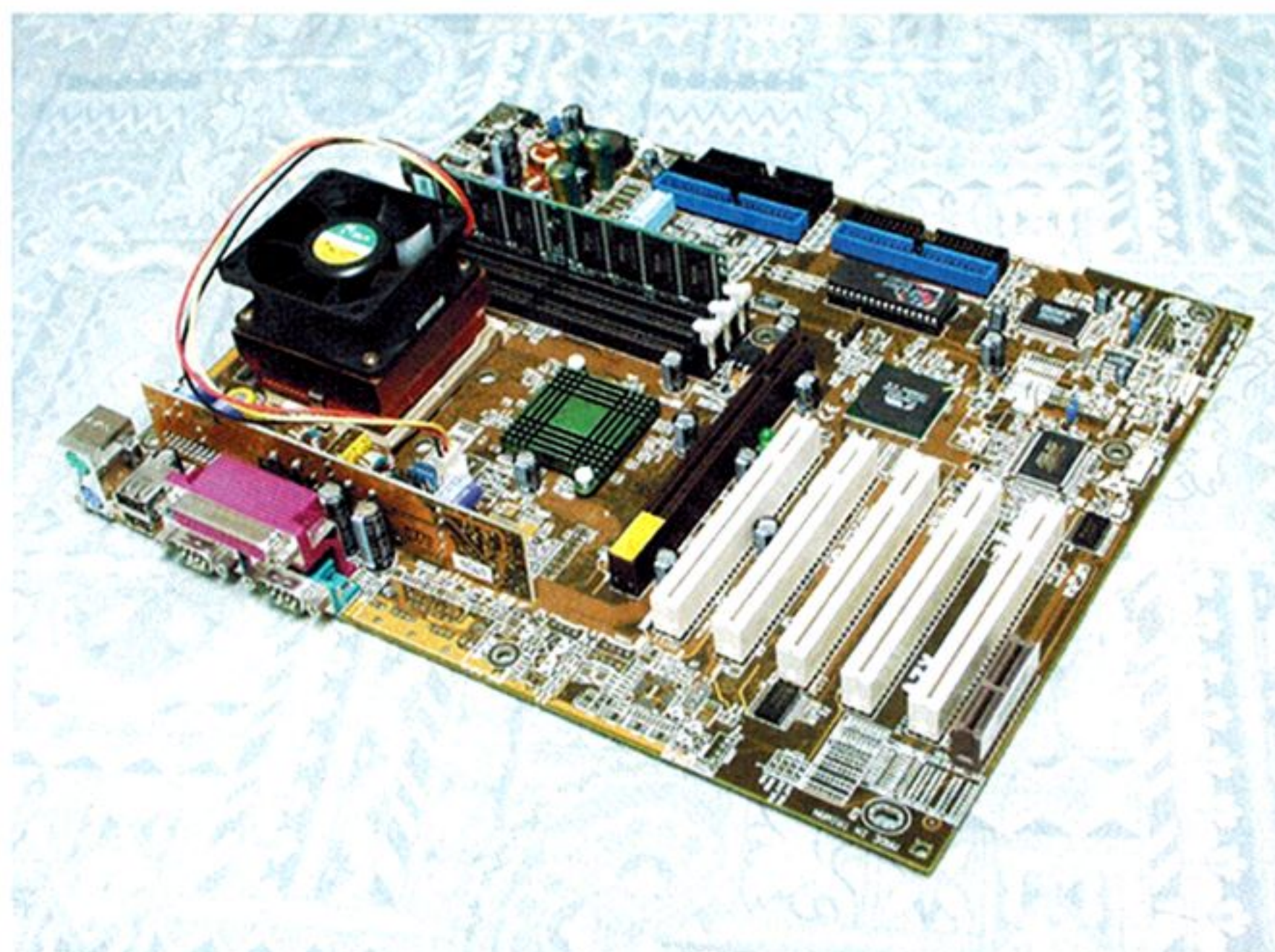
この企画も約3回目。おかげさまで，筐体の自作もすっかりと世間一般に定着してきた感じ。前回の初号機（もろもろの都合により，初回のダンボールが零号機，前回のが初号機ということになった）は，零号機以上に各方面から反響をいただいた。どうもヤツは単なる冗談ではないらしいぞ，といったところだろうか。

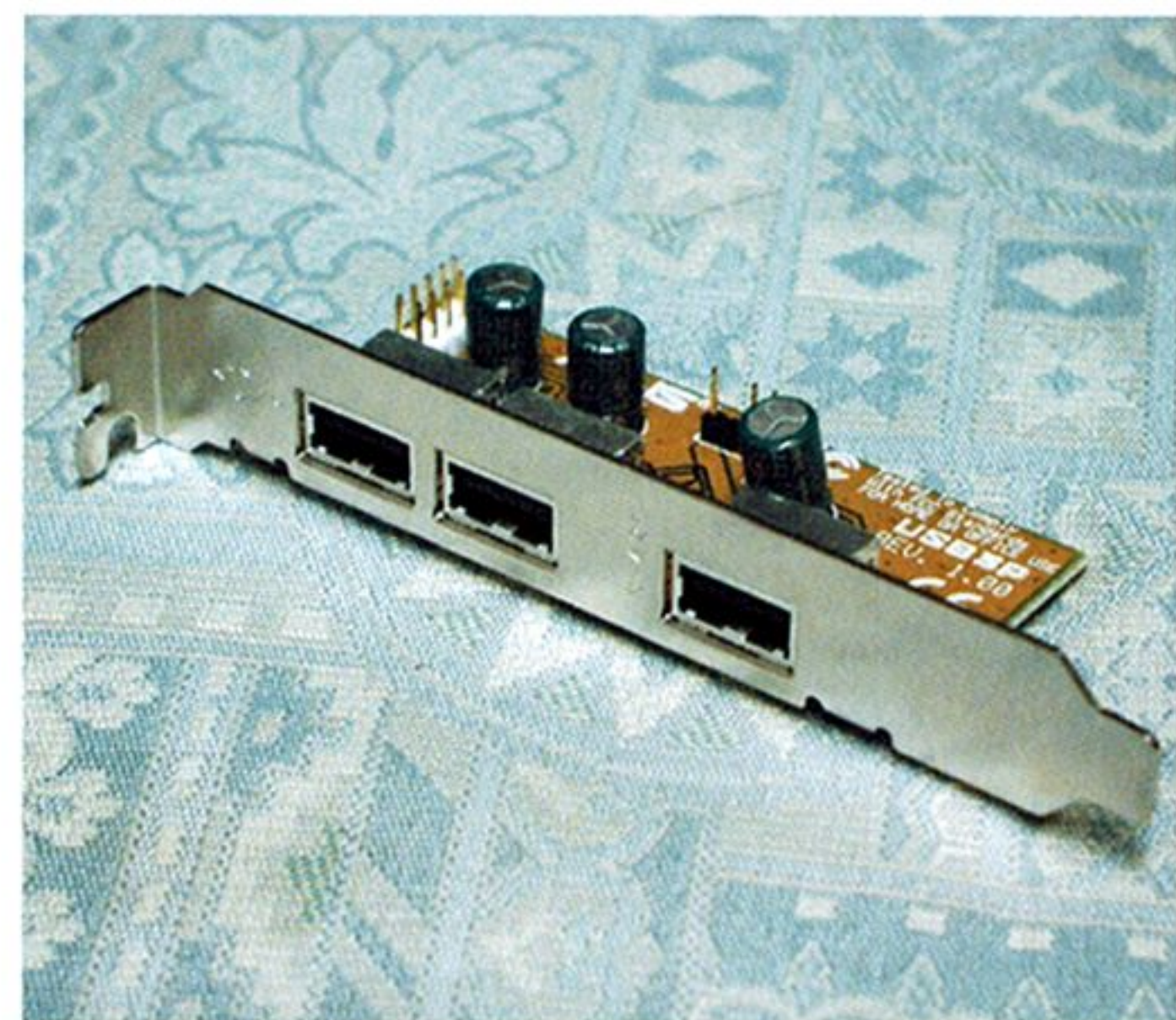
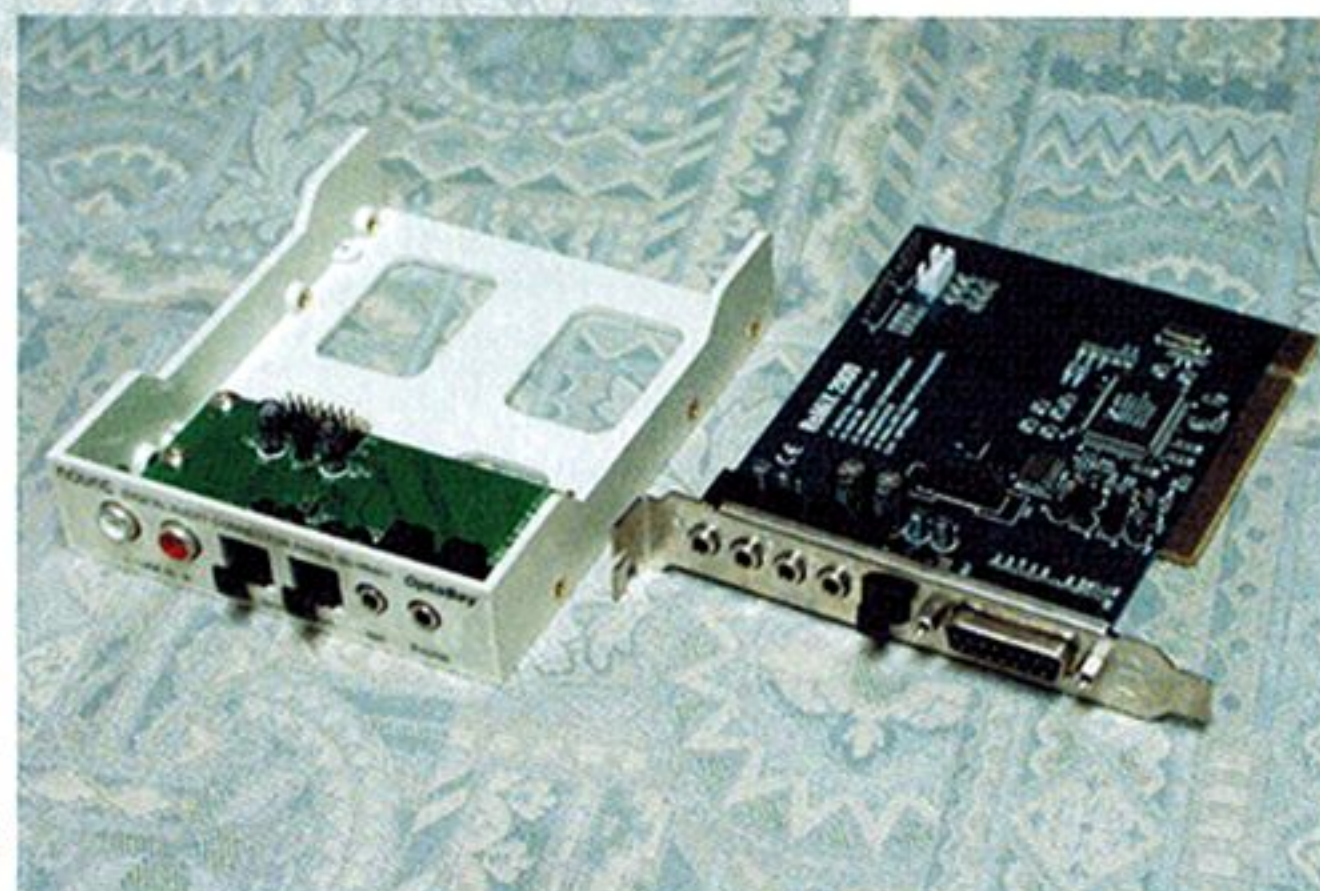
あのあと，某編集O氏から「オレにも初号機を作れー！ さもないと仕事やんないぞー！」と脅されたり，妙にFlexATXマザーの情報がいろんな人から届いたり（遠回しに自分にも作ってくれといっているらしい）。結局O氏にはカスタム機を納品したのだが，「こんなもん量産してられるかーっ！」ってことで，そこまで。同じものをいくつも作ってもしようがないので。

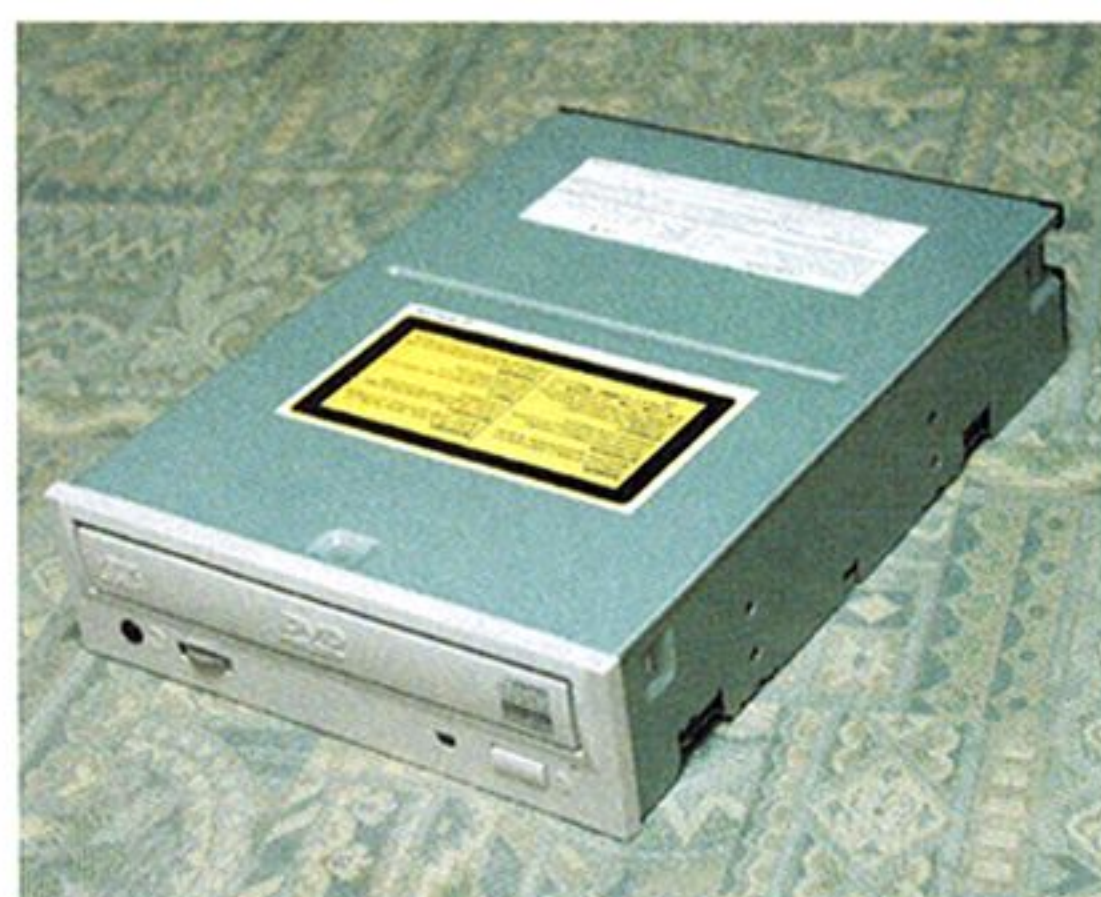
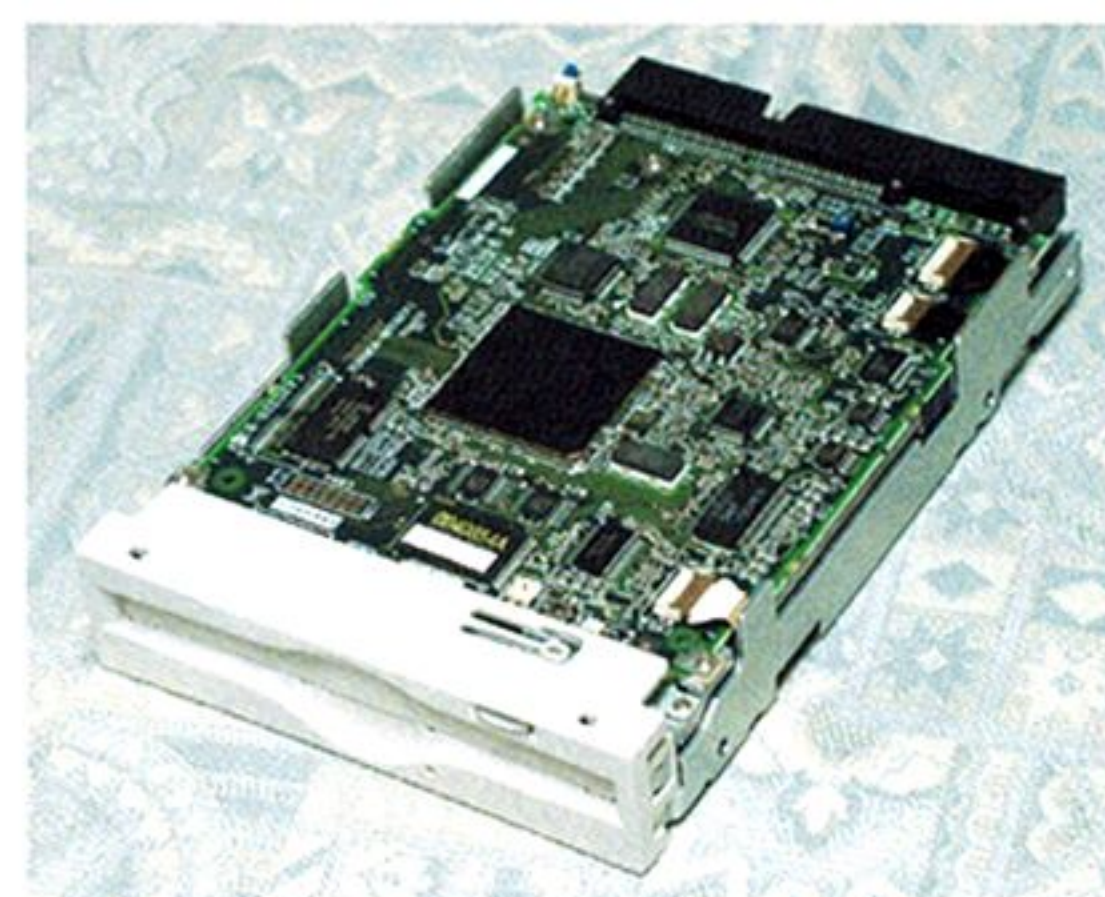
その後，某Online上ではキーボード一体型のPCを作ってみたり（ちゃんと動かなかったけど），ここんとこ自分が何屋さんなのかよくわからなくなってきたのだが，こりずに今回もメいっぱいがんばるぞ。

いつまでもパクリのデザインじゃいかんよなぁということで，ちょっと考えてみた。最初に思いついたのが，トリプルタワー（やっぱパクリじゃないのか？）。が，形状的に綺麗なプロポーションにするのがちょっと難しい。今回はちゃんとしたビデオカード積んで，ハイパフォーマンスなマシンにしたい。そこでそれを横にして，E型（三段空母型ともいう）になった。

小文字だともっとそれっぽいのだが，それはそれでミーハーなのでよしとしよう。材質は前回同様アルミ。木という話もあったのだが，ある意味アルミは木やアクリルよりも扱いやすい。せっかく買った板金曲げ機も使いたいし。そんなわけで，製作開始なのである。

E型筐体のモック。タワー部と三段部が別パーツになっている

今回はCPUにThunderbirdを採用。「ベリー，切れてナーイ」Thunderbird 700とASUS TeKの例のディップスイッチのついたA7Vの組み合わせ。でも危なそうだったので，750MHzで許してやっておいた

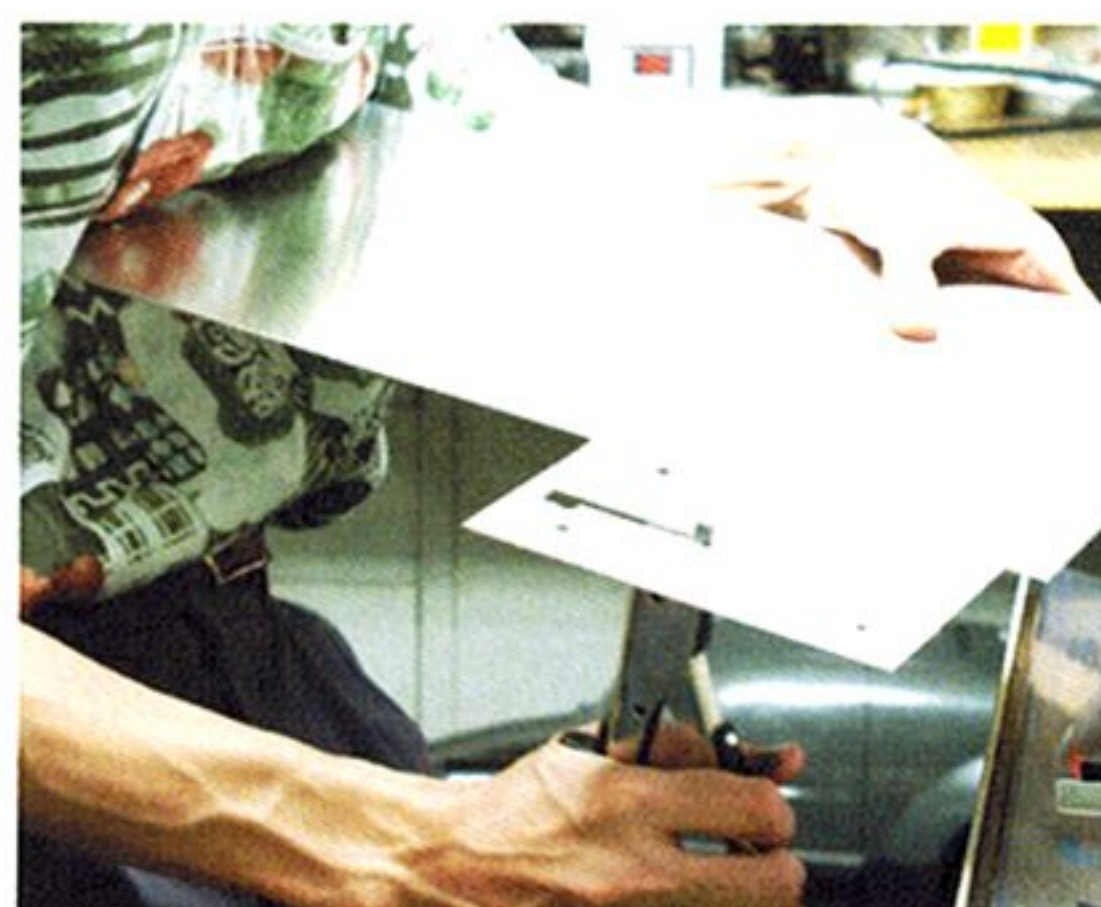
左からビデオカード（Guillemot/Hercules 3D Prophet II GTS 64MB），Ultra ATA/100 IDE RAIDカード（PROMISE/FAST TRACK 100），IEEE1394カード（恵安/KVX-100），LANカード（写真はプラネックスのFW-100TXだが，実際に搭載したのはコレガ/PCI-TXB）

A7Vは，サブボードによってUSBを3ポート外に出せるので，これをフロントにつけることにしよう

サウンドカードは，フロントにコネクタを引っ張れる（安い）ものという選択肢で，ReMiX 2000 with OptoBay

IBMの15GBプラッタ7200rpmのDTLA-307030 2発をRAIDで組む。容量は30GB×2

東芝のCD-RW/DVDコンボドライブ。DVD 4.8倍，CD-R 4倍速。コンボドライブはリコーが有名だが，あれはフロントマスクに蓋がついているので却下

富士通の640MB MO。回転数はよくわからない。GIGAMOにしようかとも思ったが，多分1GBのメディアを使うことはないだろうということで

IDE接続のPC CARDスロット。なんとなく予想はしていたのだが，ホットプラグはできない。考えてみりゃ当たり前か

今回は普通のATX電源。Athlon対応の300W

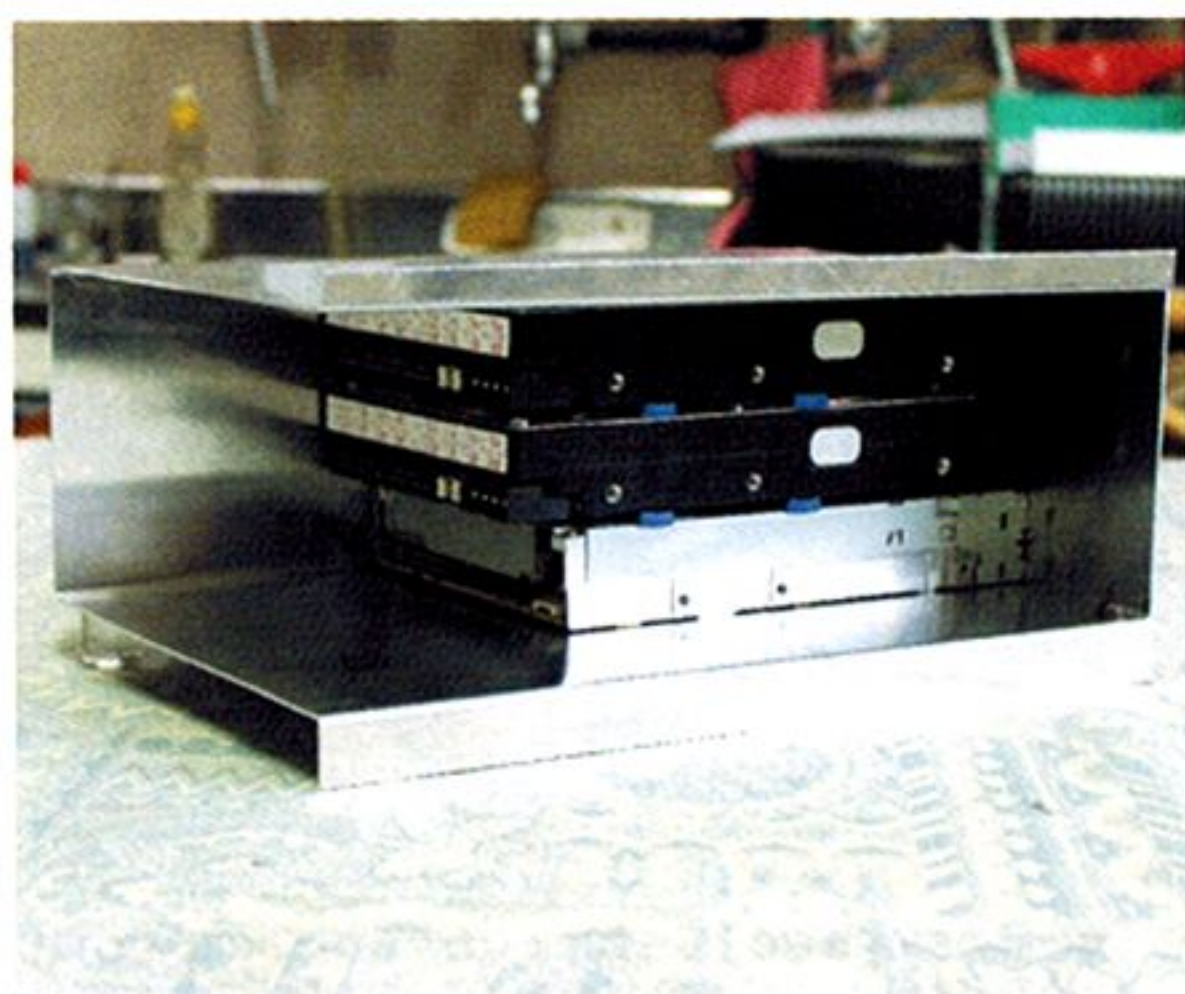
IEEE1394カードの内部ポートからフロントに出すケーブルは自作。IEEE1394の延長ケーブルというものは存在しないらしい

加工の基本，その1。ボール盤で穴開け

その2。ハンドニブラで切断

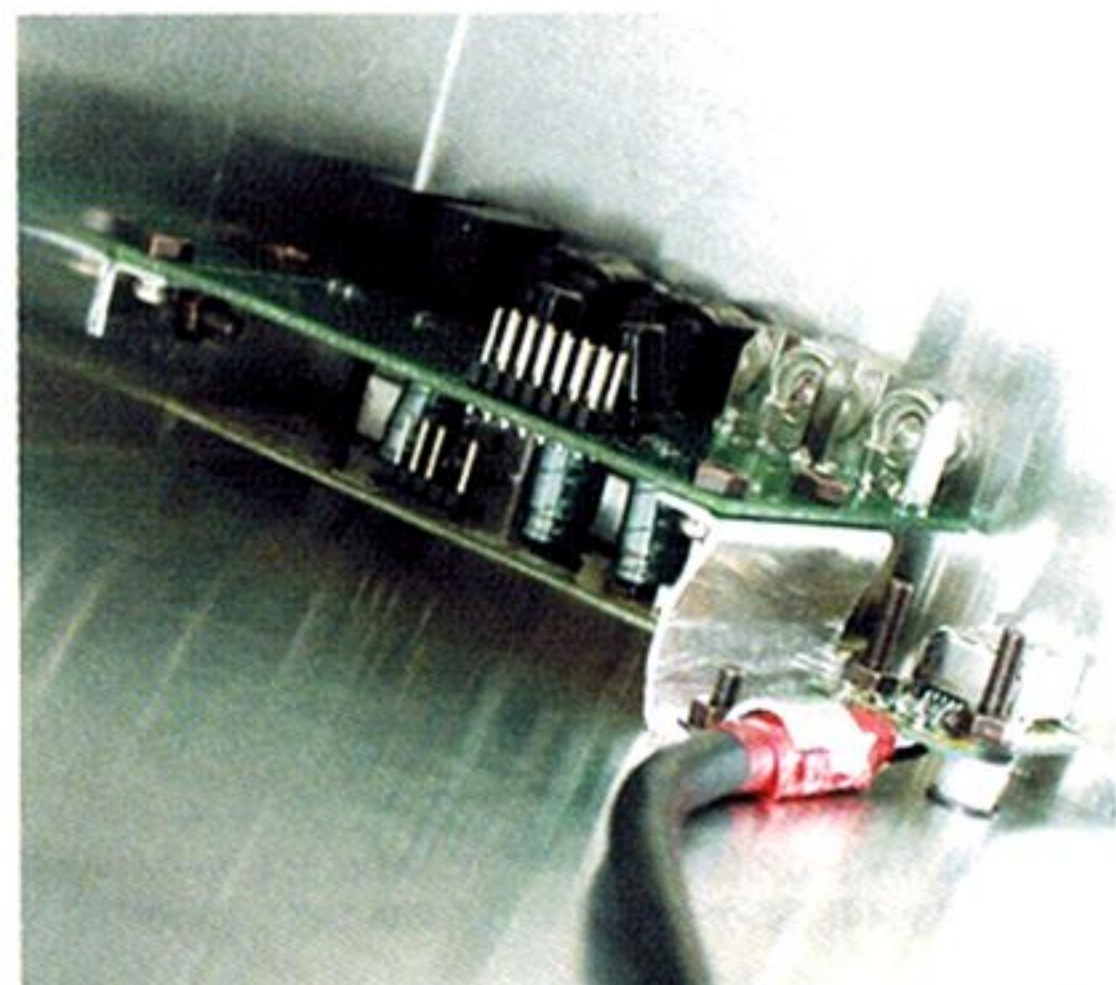
その3。必殺！　板金曲げ

タワー部の概観はこんな感じ

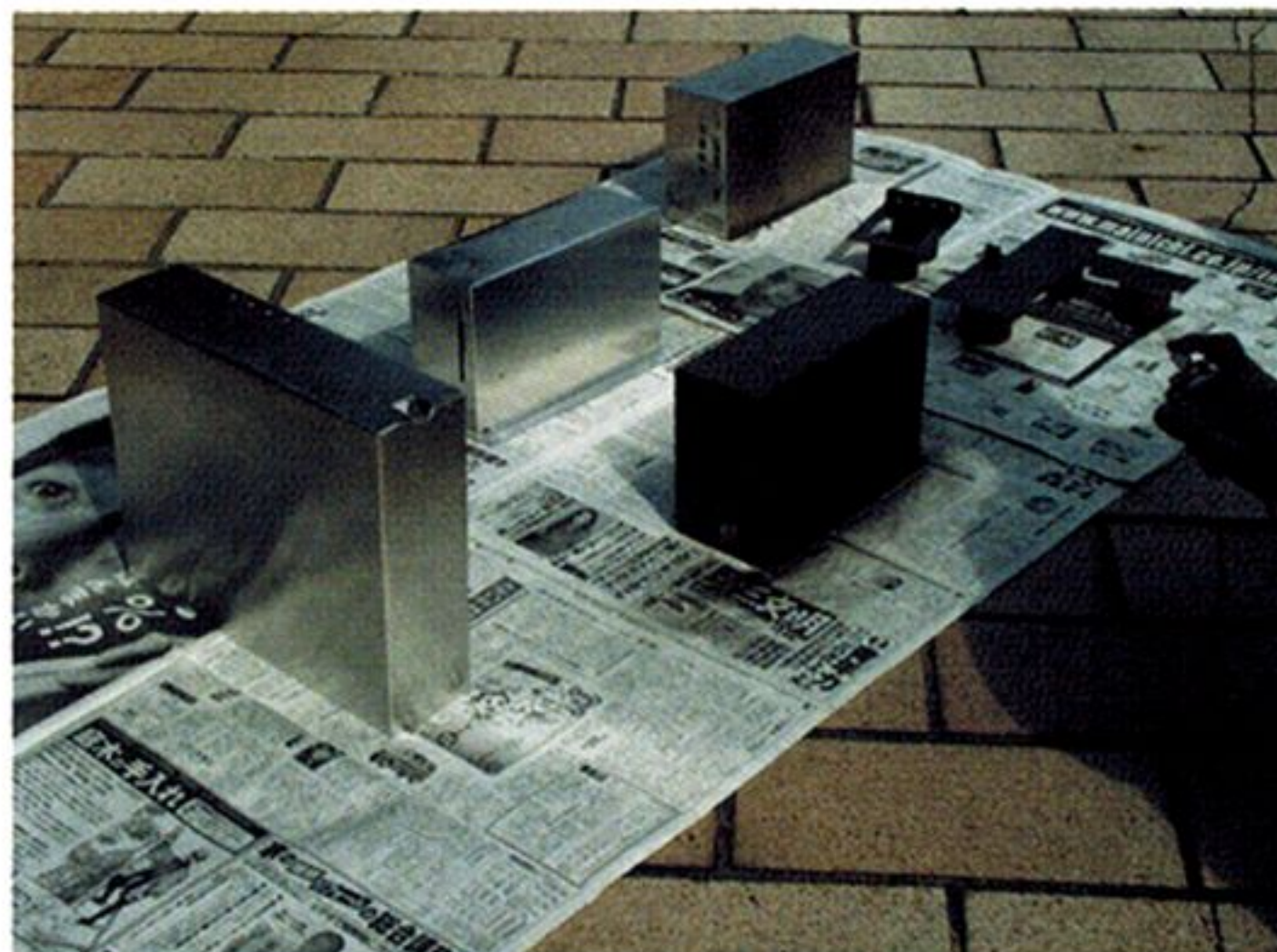
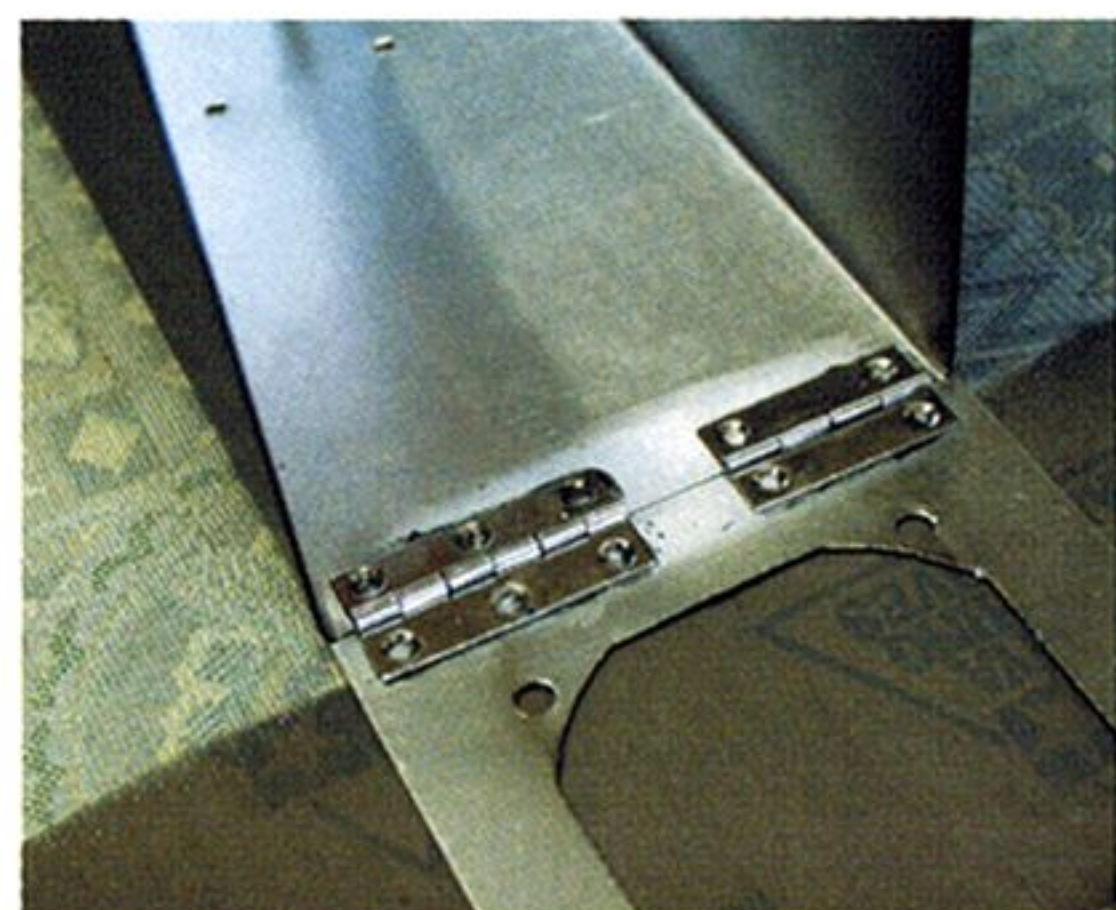
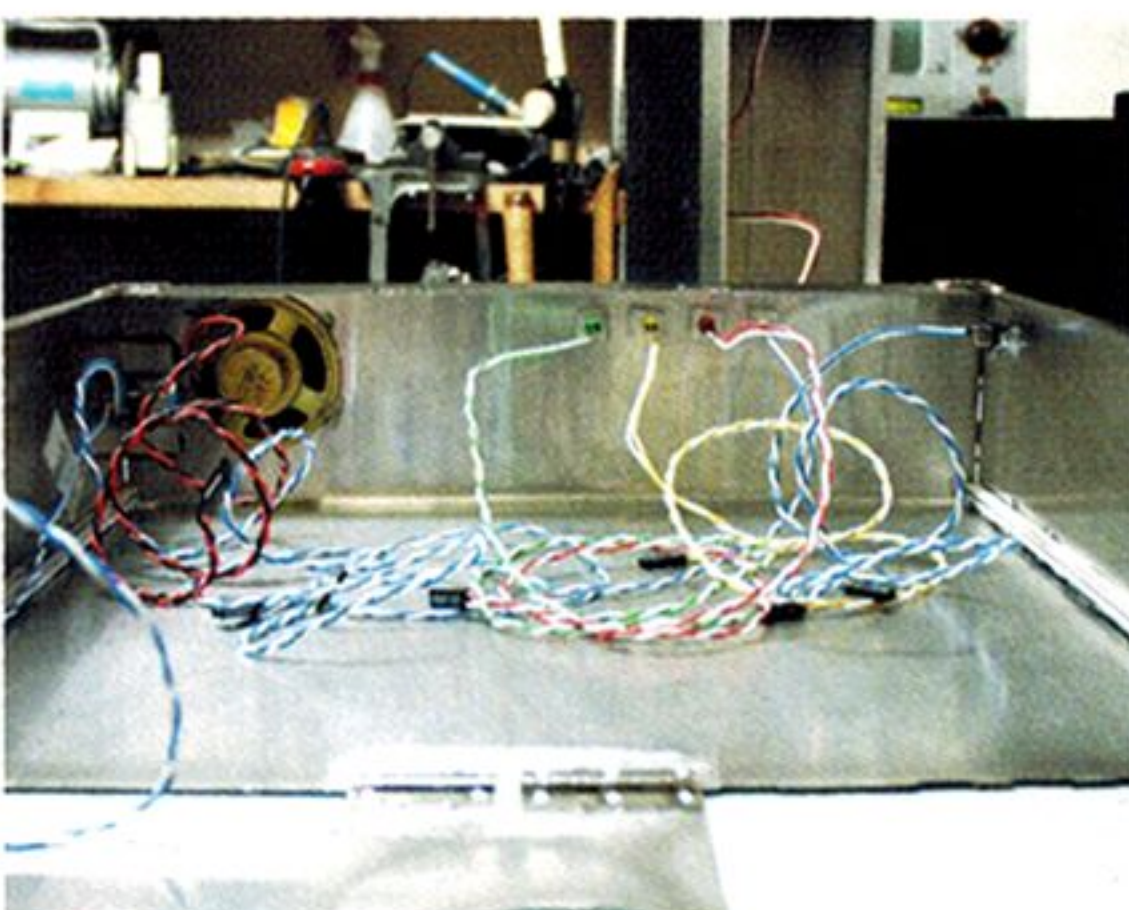
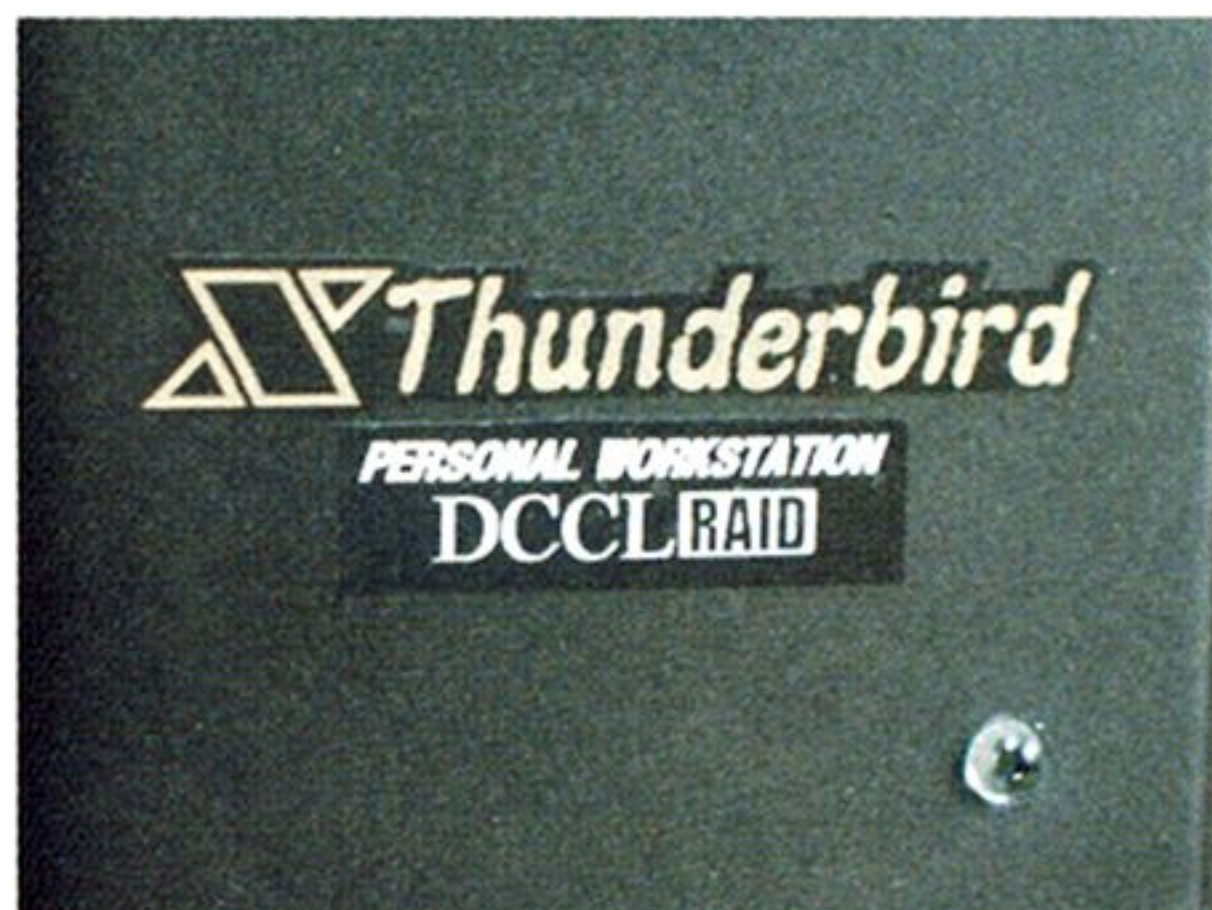
レールを上下に接着し，マザーをスライドして挿入する

上段。FDDのマスクは外して装着する

上段の内部。上2台がHDD。噛ませてある青いチップは衝撃吸収材ソルボセイン。隙間を空けることで，風を通す目的も果たす

中段。下からCD-RW/DVDコンボ，MO，PC CARD。コンボドライブとPC CARDはフロントマスクを外しているが，MOはフロントマスクに蓋がついているので，そのままとした

下段。フロントにはUSBコネクタ，IEEE1394コネクタ，音源関係のコネクタを引っ張ってある。奥は電源

下段の内部，コネクタ部分。全部で3枚の基板を固定してある

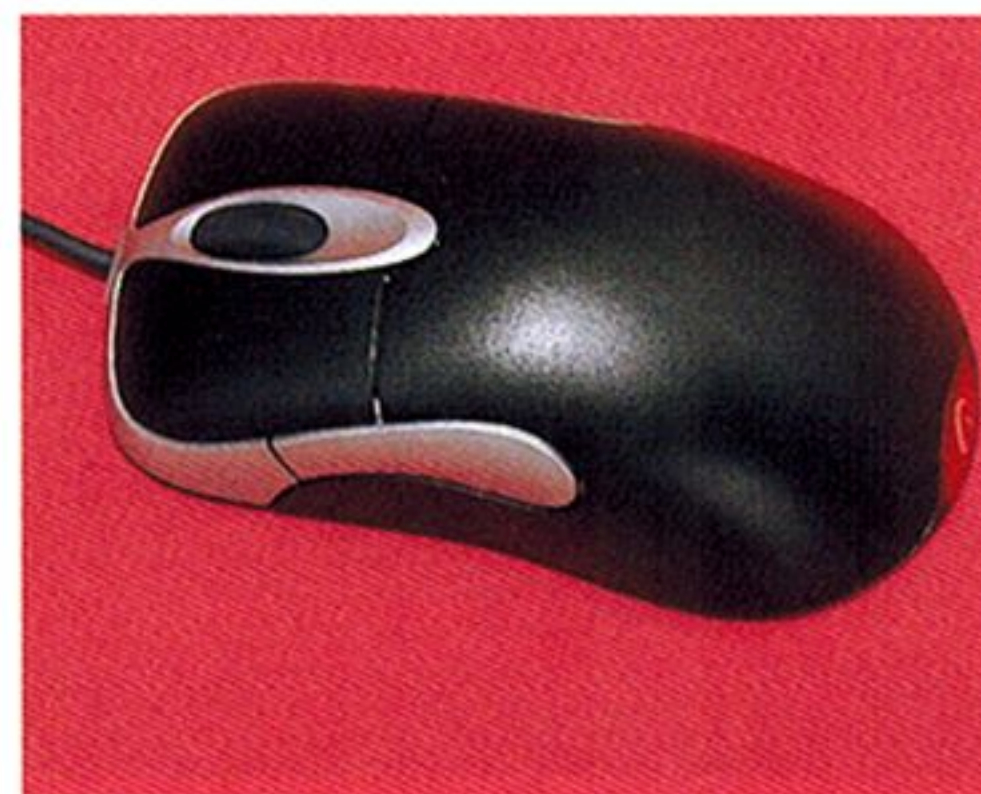
試しに合体。この段階で形状はほぼ完成

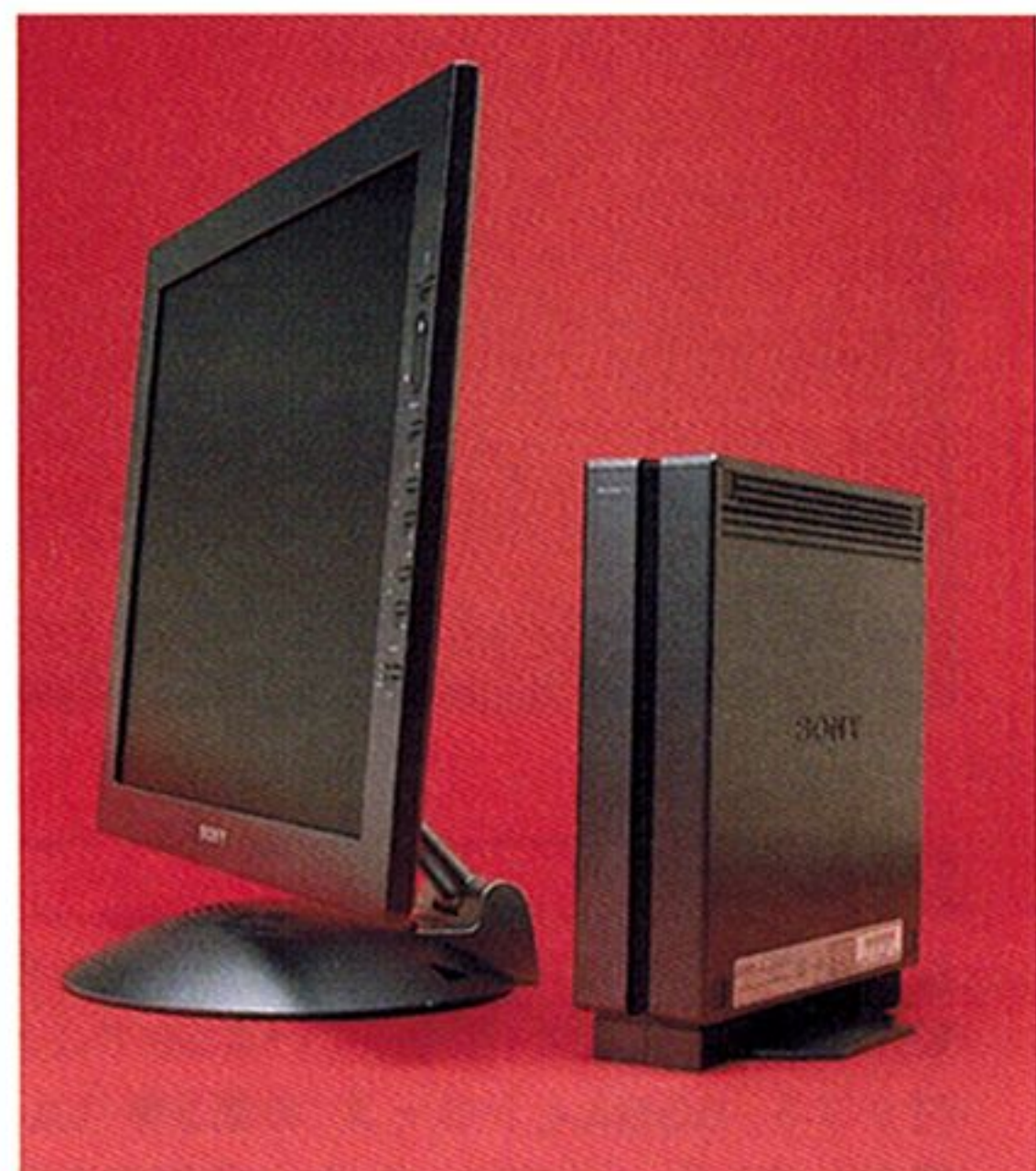
タワー下部に装着される電源スイッチ。リセットスイッチを流用した構造は前回と同じ

塗装工程へ。メタルプライマーで下地処理をして，ラッカー系模型塗料の黒鉄色も，前回と同様

蝶番の取り付けは，エポキシ系接着剤にがんばってもらうことにした

タワー部へのLEDなどの取り付け。こちらは瞬間接着剤とホットボンド

最終仕上げ。例によってMicroDryプリンタによるデカール。文字が入るとグッと締まってくる

内部の組み込みもすべて完了。当初想像していたよりも，しっかりと強度が出ている。ただし，やっぱりハンドルがないと持ちにくい

上段のフロント上部が若干寂しいが，"E"の字のプロポーションは割といい感じ

背面はまあしょうがなかろう。せめて電源背面も塗装しておけばよかったたか？

こだわりのメカニカルキーボードを塗装。元はスケルトングリーン（キートップはもともと黒）だったことが，右上のlock LEDのシールから偲ばれる。

IntelliMouse Opticalを塗装。想像していたよりも渋くてカッコよくなった。名づけてダークインテリ

組み合わせ完成之図。色が微妙に違うのと，若干横幅を取るのが珠に瑕

本体の奥行きが意外に小さいので，組み合わせるなら液晶モニタ。ってことで，SONYのチューナー付き。選択基準はボディカラー。もうちょっと暗いともっとよかったのだが

さてさて次は？

Thunderbirdの弐号機ということで，カラーはモスグリーンにして，リムーバブルフレームつけて番号振ってやろうかと思っていたのだが，カッコ悪いのでやめ。さすがに何度も作っていると，コツがわかって精度も出てきているような気がする。のだが，相変わらず設計段階ではあったはずの余裕が，実際に組んでみるとギチギチだったりする。なぜだろう？

ちなみに前回の初号機は(U)氏の私物となったが，今回は筆者の自腹である。あまり安くはないため，ちょっとヒヤヒヤものだが，一応それっぽく動いているようだ。原価に関しては，随分前からちまちまと買い集めていたので，いまとなっては随分無駄に高いものもあるが，この際コストはあまり考えていないので。さて，次はなにを組んでやろうか(まだやる気らしい)。そうそう，零号機はまだメインマシンとして現役稼動中だ(中身は多少入れ替わっているが)。

X Thunderbird DCCL RAID仕様	
CPU	Athlon 700 750MHz
メモリ	PC133 256MB
HDD	30GB×2(RAID)
チップセット	Apollo KT 133
ビデオ	GeForce2 GTS 64MB
サウンド	CMI8738
LAN	10BASE-T/100BASE TX
CD-RW/DVD	DVDx4.8/CDx24/CD-Rx4
MO	640MB
PC CARD	1スロット
IEEE1394	4ポート
サイズ	W23×D27×H31em
重量	約5kg

表1　原価表		
CPU	AMD Thunderbird 700	19,480
マザー	ASUSTeK A7V/WOA	18,800
メモリ	PC133 256MB	31,280
HDD	IBM DTLA-307030×2	35,560
CD-RW/DVD	東芝 SD-R1002	23,800
MO	富士通 MCC3064AP	25,300
PC CARDドライブ		6,980
FDD	2Mode	1,800
ビデオカード	Guillemot Hercules 3D Prophet II GTS 64MB	42,800
サウンドカード	ノバック ReMix2000 with OptoBay	6,980
RAIDカード	PROMISE FAST TRACK100	12,800
IEEE1394カード	恵安 KVX-100	5,800
LANカード	コレガ PCI-TXB	1,280
キーボード		3,480
マウス		4,780
電源		4,980
FAN		3,200
スピーカー		380
液晶ディスプレイ	SONY SDM-N50TV	138,000
ケーブル・小物類		15,000
アルミ板・加工費		10,000
塗装費		5,000
合計		417,480

DOGA-L3 概要

project team DoGA かまたゆたか

はじめに

自主制作のCGアニメを応援する当チームの活動の一環として開発・配布しているCGアニメ入門ソフト「DOGA-Lシリーズ」の第3弾「L3」が完成しました。すでに，Web上で配布を始めていますが，容量が約55MBと非常に大きいため，回線によってはダウンロードするにも，かなりの手間がかかるでしょう。そこで，本誌CD-ROMに収録するとともに，その概要について紹介します。各機能の詳細などは，PDF形式で収録されているマニュアルのほうをご覧ください。

画面1 L3のメインメニュー画面。このメニューから各ツールを起動する。L3では「アクションをデザインする」が加わっている

Lシリーズとは

まずは，L1，L2をご存じない方のために，Lシリーズを簡単に紹介しましょう。

DOGA-Lシリーズは，3DのCGアニメ作成ソフトですが，一般のCGソフトとは，開発目的や構成がかなり異なります。Lシリーズの開発目的は，まったくの初心者が，気軽にCGアニメを始めることができるような入門環境を整えるという点にあります。

昔から一般的にCGソフトは難しいといわれていますが，最近は初心者向けのソフトも出てきた半面，より高度な表現を目指して多機能化する傾向も顕著で，初心者にとっては，習得するのがなかなか困難であることに変わりはありません。数万～数十万円もするCGソフトを購入しても，ほとんど使いこなせずに挫折してしまうという話を相変わらず耳にします。このような状況では，せっかくCGに興味を持っていても，始めるのに躊躇してしまう人も少なくないでしょう。

そこで，

- 単一のソフトではなく，複数のソフトから構成されている。
- 最初のソフトは，表現力がかなり限定されていてもよいから，とにかく簡単に，すぐにアニメーションができるようにする。
- それに続くソフトは，前のソフトで習得した知識や操作をベースにしながら，少しだけ高度な機能や表現に挑戦できるようにする。
- 最終的には，市販のソフトに近い表現力を持ち，CGアニメの楽しさを知ったユーザーが，市販のCGソフトに難なく移行できるようにする。

といったシステムが必要だと考えました。つまり，初心者にとって高い壁であるCGソフトの前に，階段を用意するわけです。当チームでは，この考えをステップアップ構想と呼んでいます。

このステップアップ構想の下に開発されたのがDOGA-Lシリーズです。L1，L2，L3という3つのソフトから構成され，やさしいところから少しずつ段階的にCGアニメを習得できるような構成になっています。つまり，Lシリーズの“L”とは，“Lesson”または“Level”といった意味です。

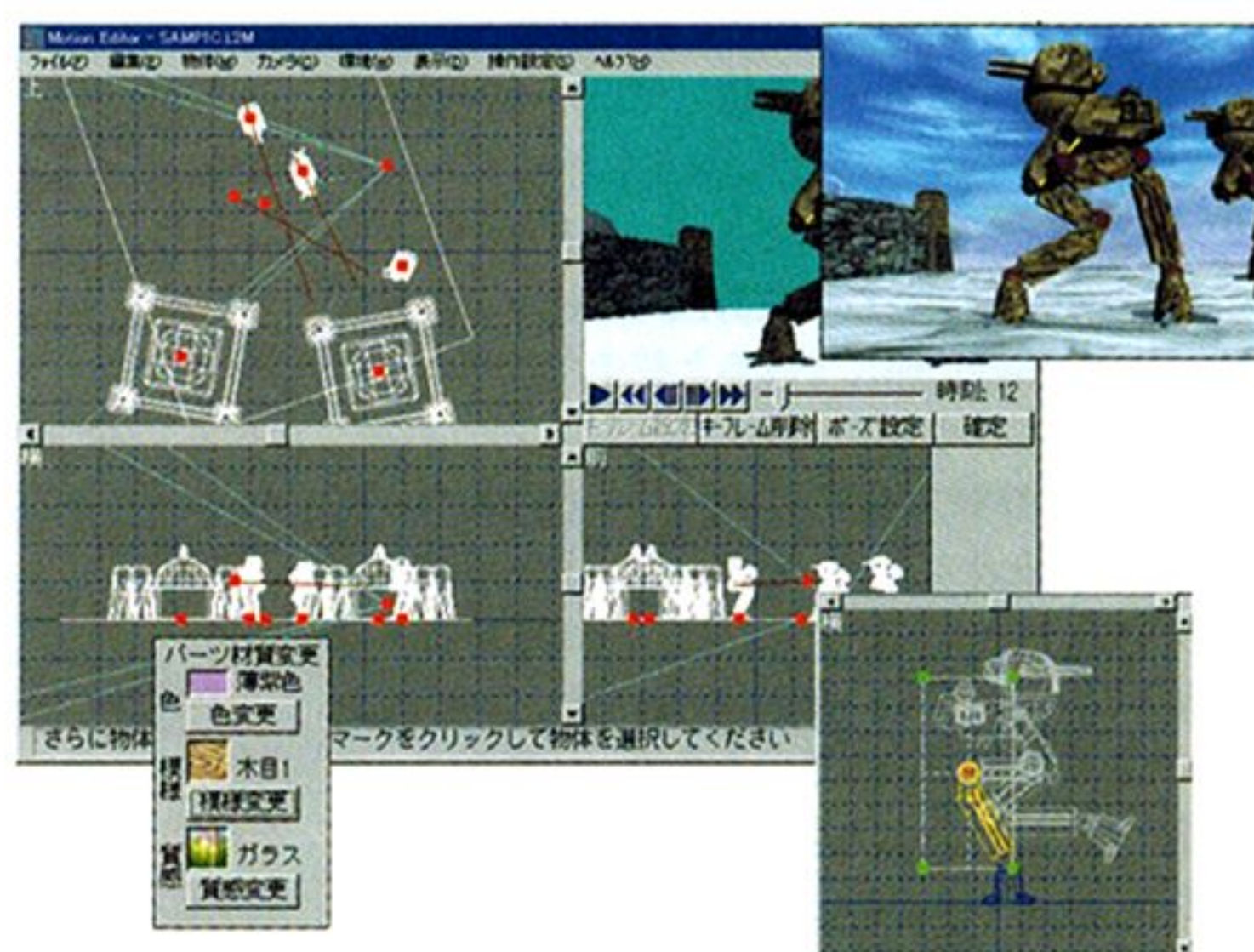
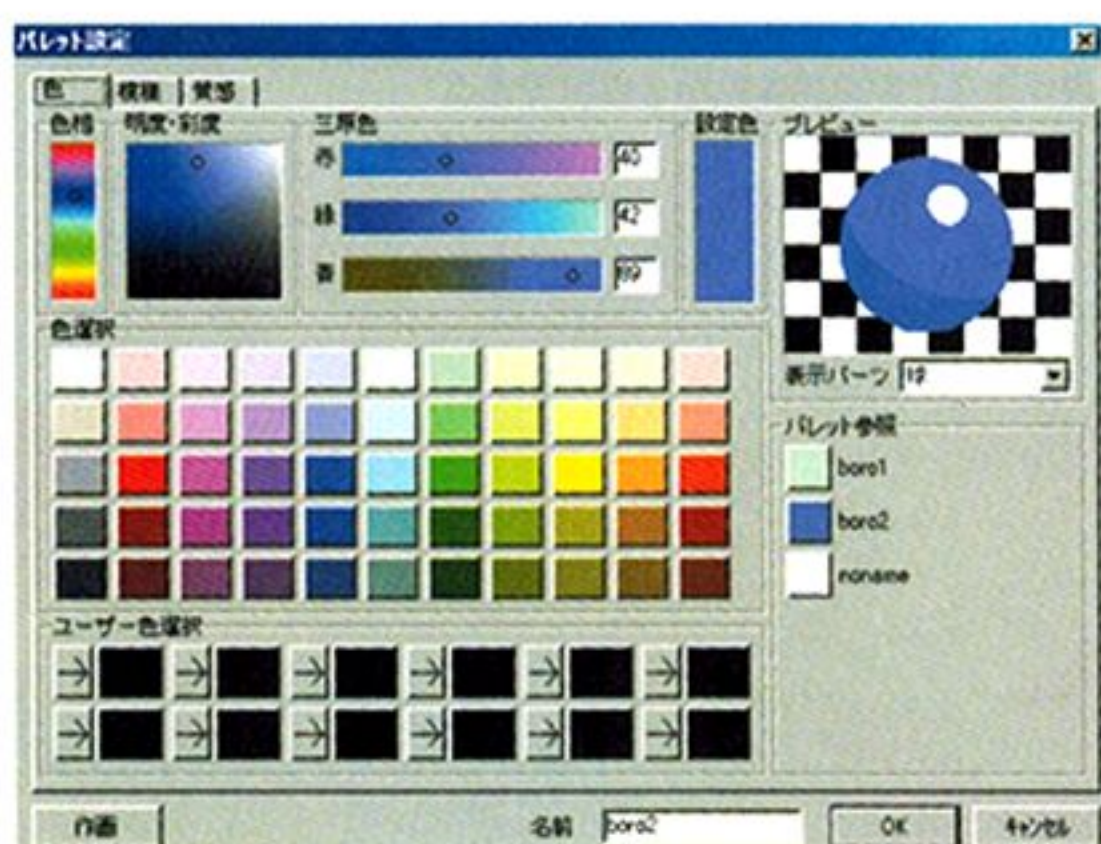
L3の位置づけ

L3は，文字どおりLシリーズのLesson3です。よく勘違いされるのですが，L3はVer.3ではありませんし，L3が出たから，L1，L2が不要になったわけではありません。初めての方は，必ずL1から順番に行ってください。

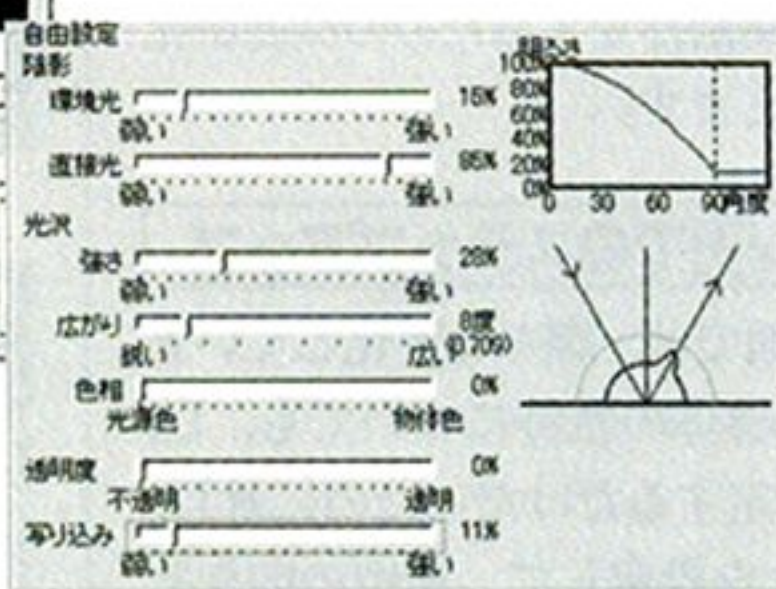
たとえると，

L1：小学校
L2：中学校
L3：高校

みたいなものです。ある町に高校が新設されたか

画面2 L1の操作画面のイメージ。形状をゼロから作るのではなく，既存のパーツを組み合わせることで，モデリングの作業が軽減されている

画面3 L2の操作画面。L2では，ロボットのような多関節物体が動かせることと，色や模様をいろいろ設定できる点がポイントになっている

画面4 色を作成する画面。色相や明度，彩度で設定したり，RGBの値を設定できる。ペイントソフトと基本的に同じだから簡単だ

画面5 模様の設定画面。凹凸や透明度を模様で設定できるし，貼る位置や大きさも少しは調節できる。その組み合わせは多様

画面7 左がリアル調の作画例。右がアニメ調の作画例。DoGAはテレビアニメの制作経験もあり，アニメ調の設定は本格的

画面6 質感設定画面。環境光，直接光などの調整については，ある程度CGの知識が必要。マニュアルにも解説がある

らといって，小・中学校が不要になるわけではありませんし，初めて学校に行く人は，必ず小学校から順番に入ります。

いきなりL3から始めようものなら，難しすぎて，なにがなんだかわからないでしょう。それにL3のマニュアルには，L1，L2と共通する機能や操作についての説明は，一切ありません。

また，登録料の面でも，

L1：無料
L2：1,000円（以前は2,000円）
L3：3,000円（ただし，L2ユーザーは2,000円）

となっています。ちょっとややこしいのですが，いきなりL3を手に入れても3,000円，L1，L2，L3と全部やっても3,000円ということです。登録料をケチるために，L2を飛ばすのは，まったく意味がありません。

モデリング面でのステップアップ

L1，L2のモデリングは，まったくゼロから作成するのではなく，事前に用意されている500種類のパーツを組み合わせるというやり方になっていますが，これはL3でも変わりません。モデリング面で，ステップアップしていくのは，形状デザインではなく，材質デザインの点です。

L1では，ほとんど材質デザインの機能がなく，物体全体に色をつけることができるだけでした。L2では，パーツごとに色，模様，質感を，用意されているものの中から選択することで，いろんな材質を表現することができました。L3ではさらに，色，模様，質感を自分で作り出すようになっています。

色は，RGBやHSVなどを使って，ペイントソフトのように任意の色が設定できます。模様は，大幅に強化されています。L2では，貼り付ける画像は模様にしかなりませんでしたが，L3では貼り付ける画像で凹凸を指定したり，部分的に光沢をつけたり，輝かしたり，透明にしたりと，質感を貼り付けることができます。事前に用意してある模様も大幅に増え，なかには燃えさかる炎のような動画も用意されています。また，ユーザーが描いた画像を貼るのも簡単になりました。

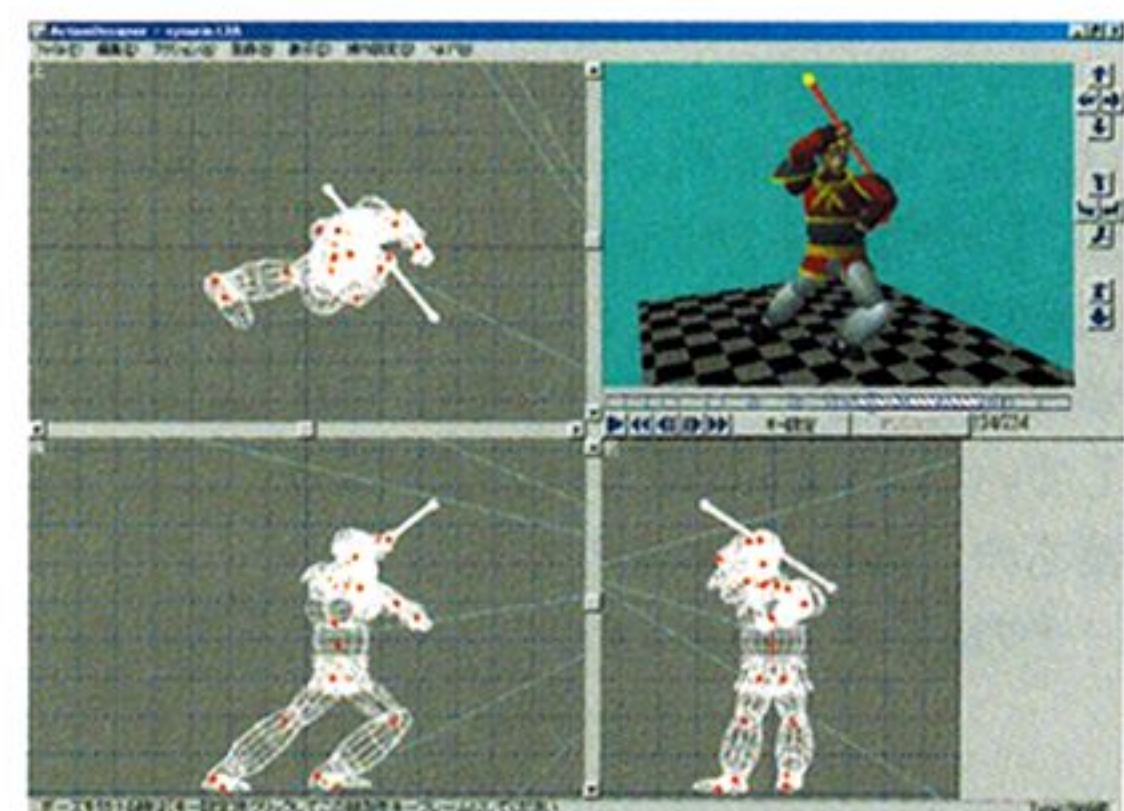
質感設定も，自分で各パラメータを直接変更して設定することができるようになっています。それ以外に，セルアニメ調の絵を作成する機能が新たに追加されました。

このように，材質設定の面では，L2とL3の差はかなり大きく，使いこなすのは若干時間を要すると思います。

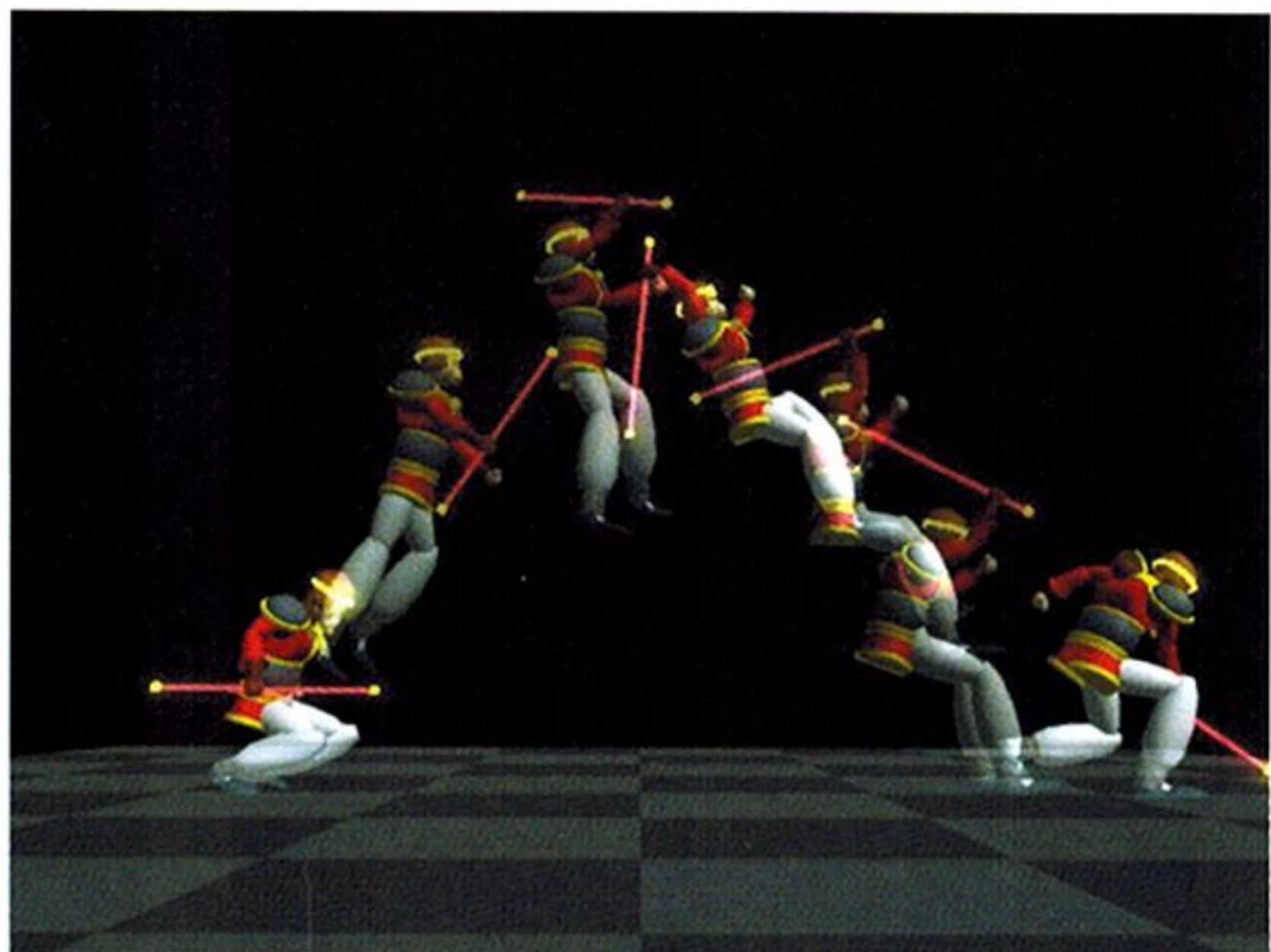
アクション面でのステップアップ

次に多関節物体の動きに注目してみましょう。L1では，そもそも多関節物体を作ることができません。L2では，多関節物体に対してキーフレームごとにポーズを設定することで，動きを設定しました。L3では，標準人体という考えが導入され，本格的に人体モデルを扱うことができます。動きも，時間情報を持った複数のポーズをひとつのデータとしてまとめ，アクションと呼んでいます。

L3では，アクションデザイナというツールが加わりました。アクション単位でのカット&ペーストができるだけでなく，それをファイルとして出力し，ほかの人体モデルに流用することができます。この機能によって，たくさんの人が動き回るようなカットの制作も，かなり軽減されるでしょう。

それから，L2で人体を歩かせるときに軸足が動いてしまい，地面を滑っているようになるという問題がありました。L3では，地面についている部分（もっとも下の部分）は自動的に固定され，動かないようにすることができます。さらに，重力も

画面9 アクションデザイナの操作画面。モーションエディタなどとよく似ているが，時間軸を操作するスライダーのあたりが異なる

画面8 よくできた人体モデルのサンプルデータがたくさん入っている。これらはすべて，L2ユーザーから提供されたもの

画面10 上向きの加速度が高いと，その瞬間自動的に接地がOFFになる。そのまま宙を飛んで，着地すると，また接地がONに

ある程度シミュレーションしていて，上方向の加速度が1Gを超える，つまりジャンプをするようなアクションをすれば，自動的に地面から足が離れ，放物線を描いて飛ぶといったこともできます。

この辺の機能も使いこなすのは難しいのですが，本格的な映像作品制作をするための機能が充実してきているといえます。

シーンデザイン面でのステップアップ

シーンデザインの面では，細々と変化しています。まず，移動する物体の軌跡についても，L1では始点と終点を指定するだけだったのに対し，L2ではキーフレームを設定して，途中の向きや大きさに変化をつけることができるようになりました。L3では，さらに通過点が加わったことで，相当複雑な軌跡を描くことができます。

光源に注目すれば，L1では白い平行光源がひとつだけでしたが，L2では光源に色を設定できるようになりました。そしてL3では，平行光源のほかに，点光源，スポット光源が加わり，それらを自由に移動することができます。色についても，キーフレームごとに色を設定したり，点滅やロウソクの炎のように不安定な光も表現できるようになっています。

カメラや速度の設定なども，少しずつ設定項目が増えていますし，L1，L2になかった表現としては，レンズフレアなどがあります。

その他

これは当初の計画にはなかったことですが，L3はレンダラが新しくなっています。専門的にいえば，L1，L2ではグローシェーディングだったのが，L3ではフォンになりました。その結果，作画速度は少し遅くなりましたが，作画クオリティはかなりよくなっています。また，L1，L2で発生していた模様が歪むといった現象もなくなり，影落ちの表現も可能になりました。

その他，モーションの途中でユニットをいろいろ差し替える切り替え表示，1本の木から森を生成する大量複製，背景の作成に便利な複数物体のカット＆ペーストなど，便利な機能もいろいろ用意されています。

Lシリーズの今後

L3は，まだバグも多く，一部の仕様も改善したいと考えています。同様にL1，L2も，ユーザーの皆さんに発見していただいたバグを修正した最新版を作成し，ともに昨年の冬のコミケで発表しました。ちょうどそこで，世紀も変わったことですし，とりあえずLシリーズはいったんおしまいにして，今世紀は別のシリーズを企画しています。

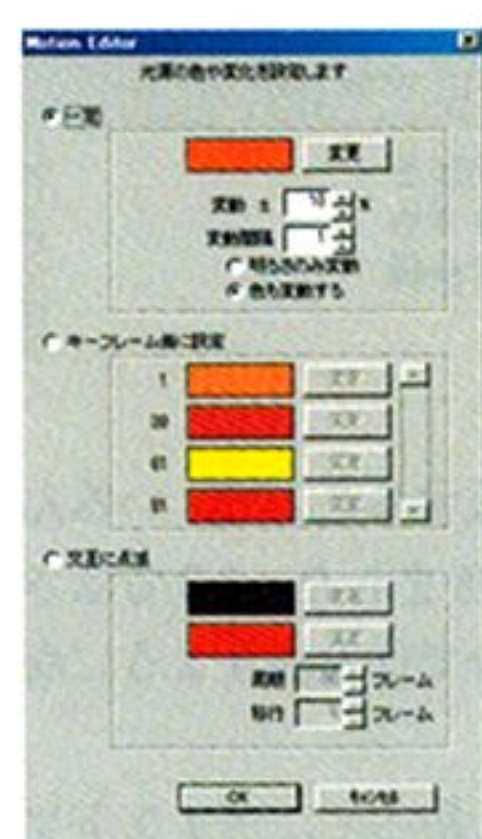
当初の計画では，LシリーズはL5まで計画されていました。しかしながら，現状では，L4，L5を開発する必要性をあまり感じません。まずL4では本格的なモデリングをする予定でしたが，これはもうメタセコイアや六角大王という優れたモデラが存在しますし，フリーで配布されています。L3では，メタセコイアや六角大王で制作した形状をコンバートする「objcomv」というツールが一緒に配布されていますので，もう事実上L4が存在するようなものです。

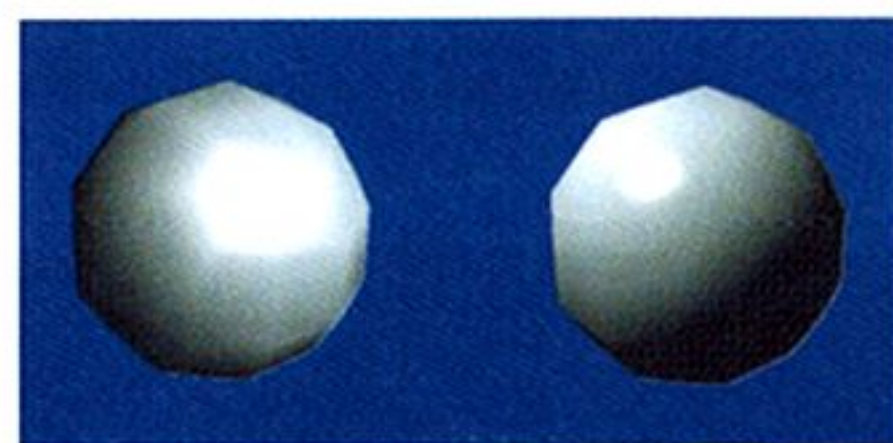
L5では，市販の本格的CGソフトに近いレベルのソフトにする予定でしたが，L3が当初の予定より高機能すぎて，多くのユーザーが使いこなすのに苦戦しているようです。ですから，L3やメタセコイアを使いこなせる実力があれば，もう市販CGソフトに移行しても問題ないでしょう。

それから，音声やテロップを入れる映像編集ソフトが未完成のまま残っています。しかし，本格的に作品制作する方ならビデオキャプチャカードぐらいは購入するでしょう。そうすれば，たいていの場合，プレミアなどの映像編集ソフトがついてきます。

このように，初心者が気軽にCGアニメを始めることができるような入門環境を整えるというLシリーズの目的は，L1～L3によって，だいたい達成できたように思います。現に，多くの教育機関などでLシリーズが使われ，好評を得ています。また，Lシリーズをパッケージソフトとして販売したいという話も進んでいます。当チームとしては，いろんなメディアを通じて，Lシリーズが配布され，多くのCG入門者を増やすことができれば，幸いです。

さて，次のシリーズですが，その内容はまだ秘密です。今年の夏か秋ぐらいには，公になるでしょう。そのときは，皆さんをあっといわせることができるように，頑張りたいと思います。

画面11 通過点を複数設定し，その位置での方向を決めることができるので，急に折れ曲がるなどの複雑な軌跡を表現できる

画面12 光源色の指定画面。時間とともに色が変わったり，焚き木の炎のように，色や明るさを変動させることも可能になった

画面13 カメラの視界に強い光源があるときに発生するレンズフレアも，控えめ，普通，派手の3種類が用意されている

画面14 左がL2で描いた球。右がL3。L3は，レンダラが新しくなっており，ちゃんとハイライトが丸く表示されている

画面15 L2ではできなかった影落ちがL3では可能に。しかし，たくさんの物体に影をつけると作画速度はかなり落ちてしまう

GraphicLaboratory【番外編】

BGMジェネレータ試用レポート

「自動作曲はCGアニメの救世主になるか？」

project team DoGA かまたゆたか

CGアニメ制作における問題点のひとつとしてBGMがある。既成の曲を使えば著作権法に触れるし，かといって誰にでも作曲できるわけではない。映像のイメージにあって，ぴったりの長さのBGMをいかにして得るか？　そこで今回は，2000年7月20日に松下電器から発売された自動作曲装置「BGMジェネレータ」を紹介しよう。

BGMジェネレータ概要

「BGMジェネレータ」(以下BGM-Gと略)は，20cm四方ほどのハードと，CD-ROMで供給されるソフトから構成される。パソコンへの接続は簡単で，USB端子をつなぐだけだ。ハード側のスイッチ類は音量しかない。電源を入れると，正面のマークが青い発光ダイオードで光り，かっこいい。

ソフトのほうも少しも難しくない。基本的な使い方は，120種類あるリズムと60種類あるメロディのなかから好きなように選択して，曲の長さを指定するだけ。その他，曲をアレンジするパラメータとして，曲調の「明るい」「暗い」や，曲のにぎやかさなどが設定できる。そして，「ソング作成」ボタンを押せば，1曲できあがり。

次に，できた曲の編集画面になる。ここでは，フレーズ単位でバリエーションを変えたり，演奏する楽器を減らすといった操作ができる。

これで曲は完成だが，やろうと思えばいったんMIDIに出力し，そのデータをさらに細かく設定するといった機能も備えている。

さて，なんといってもいちばん大切なのは，出力される曲のクオリティだが，これはなかなかたいしたものである。自動的に作成したとは思えない，破綻のない曲で，自主制作のCGアニメ作品のBGMとしての実用性にはまったく問題ない。カタログなどには「プロのCM制作にも」とあるが，これもあながち無茶とはいえない(本誌付録CD-ROMオーディオトラックにサンプル曲を収録)。

それもそのはずで，厳密にいえばこのBGM-Gは自動作曲していない。あらかじめ用意された60曲を，いろんなリズムでアレンジしたり，小節単

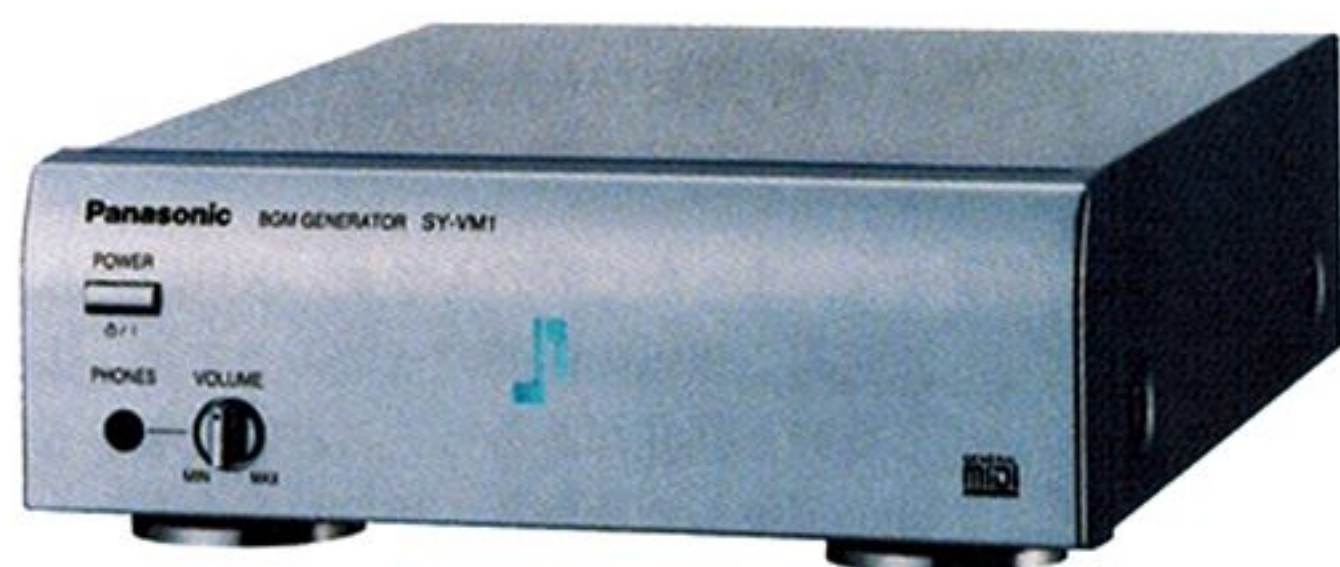
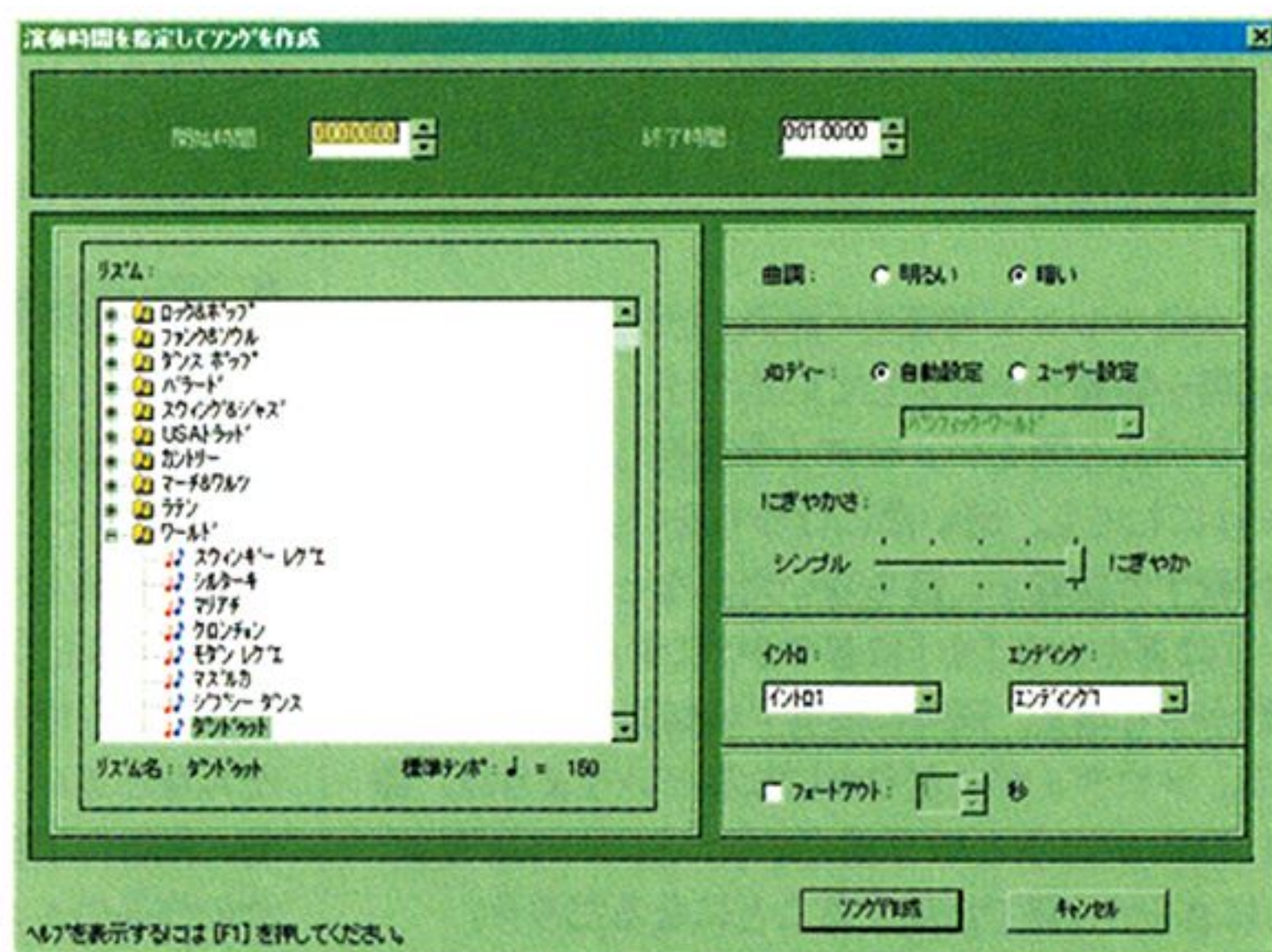
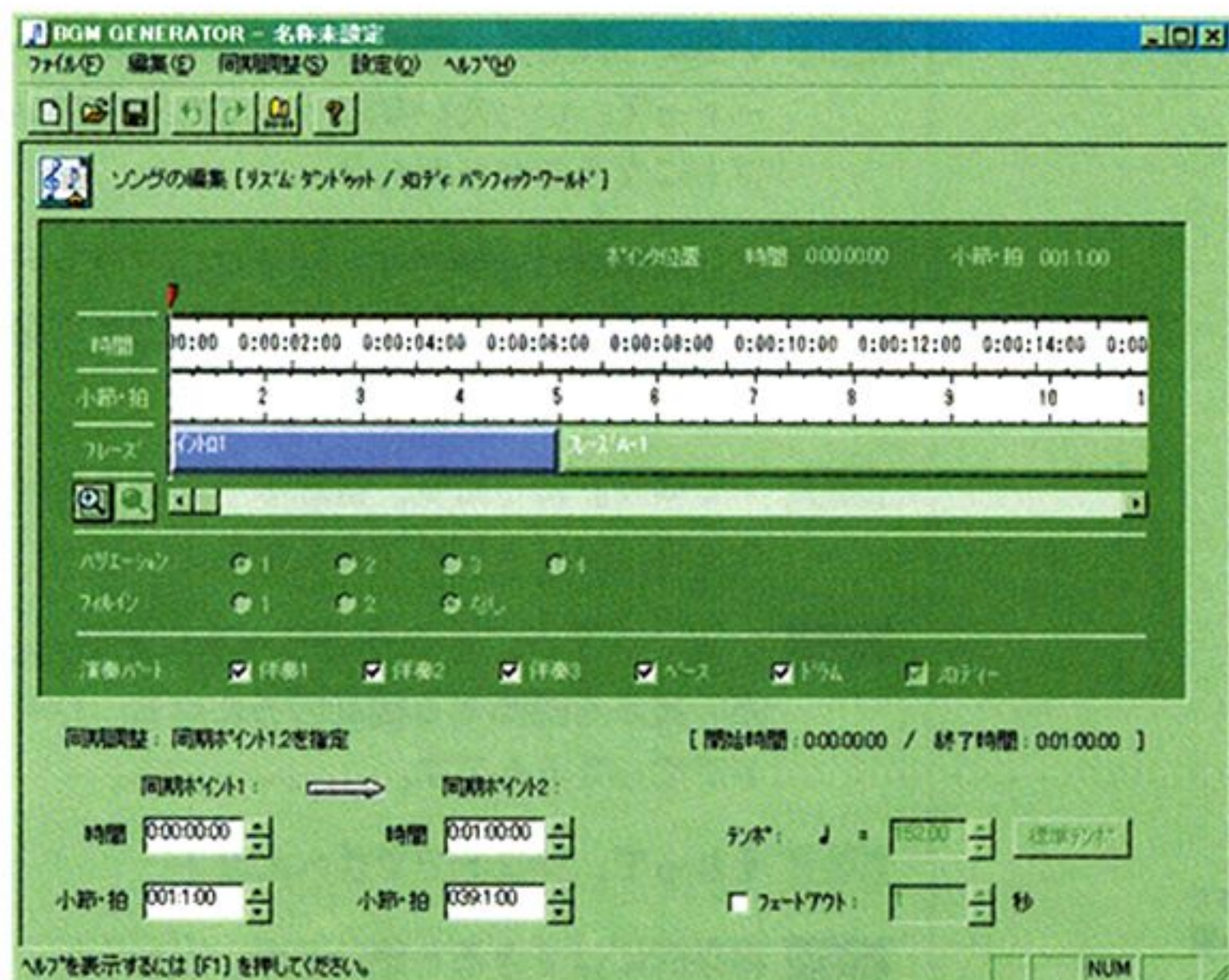
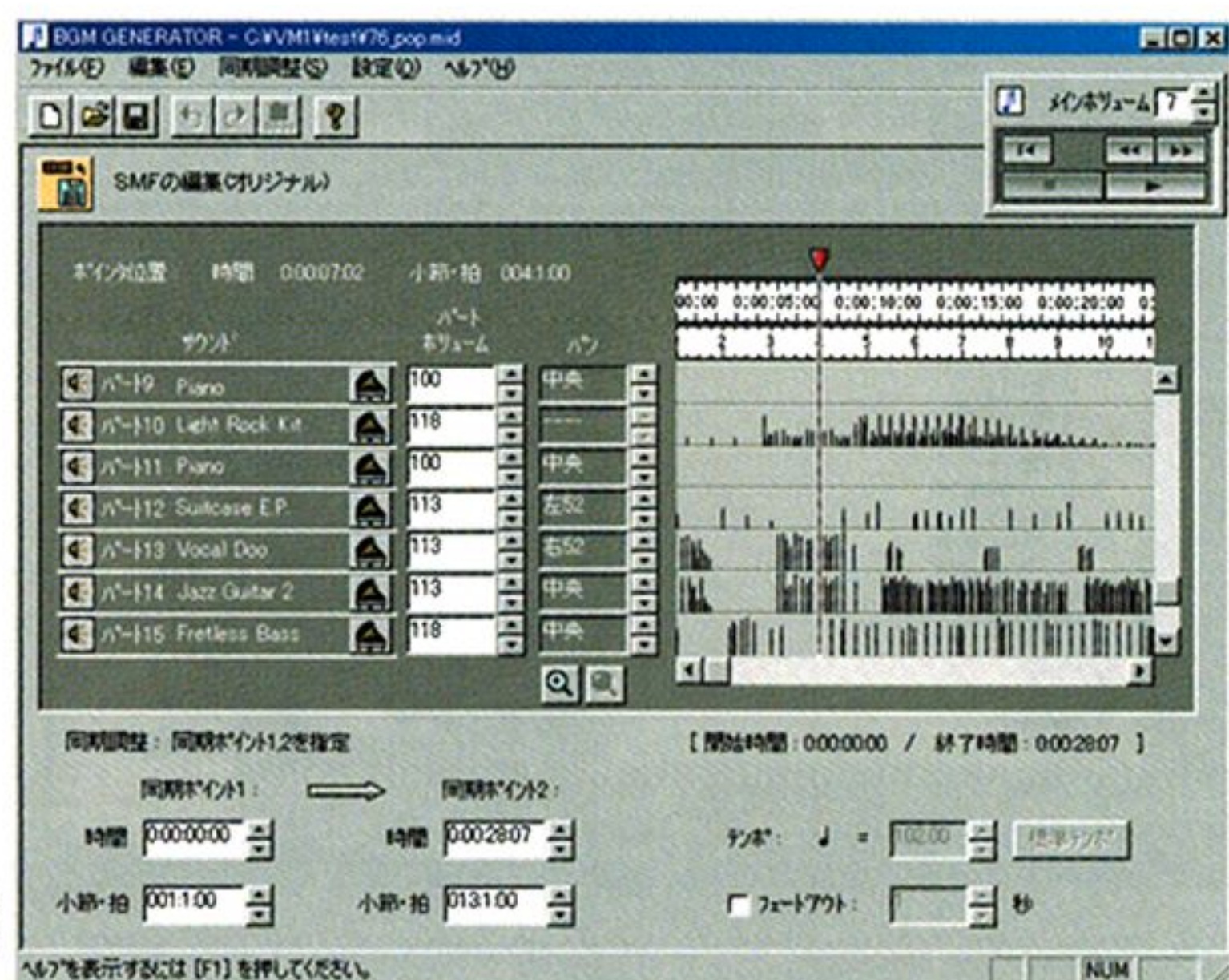



写真1 これがBGMジェネレータ

画面1 大まかなパラメータを設定して

画面2 フレーズごとの調整もできる

画面3 MIDIに出力してさらに細かく調整

位で演奏時間を調節しているだけである。

以上でお値段は79,800円。アマチュアでもちょっと無理すれば，十分手の出る価格帯ではないだろうか。

BGM-G使用感

さて，各機能をもう少し細かく見てみよう。まず画面の左半分を占めるリズムだが，3/4拍子といった分類ではなく，ポップス，ロック，バラード，ジャズといった音楽のジャンルが階層的に分類されている。つまり，リズムを選択するというよりは，ここで曲のイメージを設定する感じだ。

これが120種類もあるというのが，このBGM-Gの売りでもあり，最大の欠点でもあるように思う。なぜなら，音楽の専門家ならともかく一般の人は，音楽のジャンルを120種類も知らない。だから，どのジャンルがどんなイメージなのか，さっぱり見当もつかないのだ。「シルターキ」とか「マズルカ」って，皆さん知ってます？　120種類を順番に聞くだけでも大変と思うのだが，使っているうちに覚えていくものなのだろうか？

もうひとつの大きな要素であるメロディの選択だが，こちらはBGM-G用に作曲された曲と，著作権上問題のない有名な曲（蛍の光，ジングルベル，森へ行きましょう）とが半々ぐらいで収録されている。

有名な曲のほうはともかく，新たに作曲された曲は，「ロック1」とか「バラード4」とか表示されているだけなので，どんな曲なのかは聴いてみないとわからない。この辺は，リズムと同じような問題点を感じる。

次にアレンジ関係の「曲調」だが，これは単に長調か短調かを設定しているだけ。だから，明るいリズムと明るいメロディを選択し，曲調を「暗い」にしても，暗い曲にはならない。「明るい」と比較すれば確かに落ち着いたかなという程度だ。

「にぎやかさ」を「にぎやか」にすると，音符の数が少し増え，同時に演奏する楽器の種類が増える。メロディがにぎやかになるというよりは，曲に厚みが出るという感じ。たいていの場合は，「にぎやか」側で使用することになるだろう。

「イントロ」と「エンディング」は，それぞれ4パターン用意されており，またその長さも調節できる。

以上のように，作成される曲は，リズムとメロディの選択だけでほぼ決まり，その他のアレンジはそんなに大きな影響を与えないという印象だった。いったんMIDIにでも落とさない限り，ユーザー側で大胆にアレンジするというのはほとんどできない。

しかしながら，たとえばメロディとして「蛍の光」を選択し，リズムに「バラード」を選択すれば，実に哀愁を帯びた「蛍の光」になる。そして，リズムを「マーチ」に変更するだけで，思わず歩き出したくなるような明るく楽しい行進曲になる。それだけで，十分すごいと思う。

それに，この装置の最大のポイントは，映像の都合にあわせて，たとえば“1分17秒”というように時間を指定してやれば，ぴったりの長さで演奏が終わるように自動的に編集してくれるという点だ。実際いろんな長さの曲を作ってみたところ，場合によっては，やや強引に終わらせたという感じの曲になることもあったが，全体的にはなかなかうまく処理していた。秒単位まで正確に指定できるのは，映像作品を制作する者にとって，実にありがたいといえるだろう。

なお，曲の長さを指定しないモードもある。こちらでは，曲をずっと流しながらインタラクティブにリズムやメロディの変更ができる。つまり，このモードで自分のイメージにあう曲になるように試行錯誤し，最後に長さを指定するという使い方を想定しているようだ。

開発秘話

実はこのBGM-Gを開発者した松下電器電子楽器事業部の小谷博之君は，元DoGAのスタッフだ。元といっても，現在でもCGAコンテストで作曲を担当してくれている。

毎年CGAコンテストに入選する作品のいくつかは，著作権上問題のあるBGMを使用している。そんなときは，当方でその作品用のBGMを作曲し，作者の了解を得たうえで差し替えているのだ。その作品のイメージにあって，作品の長さにぴったりの曲を作らないといけない……そう，まさにBGM-Gは，CGAコンテストのために生まれてきたような装置なのだ。

そういえば前々から小谷君に提案していた。事前に用意されたサンプル曲を小節単位で編集して，作曲できない人でも，自分の作品の長さにあった曲を作れるようなツールを開発して，DoGAから発表しないかと。ということは，もしかしてこのBGM-Gって，私のアイデアを盗んで開発したんじゃないのか？　ちょっと，小谷君を呼んで，問いただしてみよう（笑）。

かま BGM-Gの発売おめでとう。

小谷 おかげさまで，従来になかった斬新な商品とご好評をいただいています。

かま で，だいぶ前からああいうソフトを開発しようって，いってたやん。

小谷 そうですね。

かま 私のアイデアをそのまま商品化したんとちゃうんかいな。

小谷 そんなことないですよ。偶然ですよ。

かま お前さんが企画したんやったら，偶然ってことはないやろ（笑）。

小谷 ほらっ私，最初は音楽と全然関係のない事業部に配属されたでしょ。それが社内の志願制度を利用して，やっと電子楽器のほうに移動できたんですよ。そして，さぁなにをやるんだということになったとき，電子楽器のほうで挙がっていた企画の中に，これがあったんですよ。

かま なんや，そしたら企画は小谷君やなかったんか。

小谷 えぇ，まったく別の人です。事業部でもまだ企画の候補の段階だったのですが，私もこういう需要があるってことは身をもって知ってましたから，ぜひやろうということで担当になりました。

執拗な追及

かま いろいろ聞きたいねんけど，まず，このハードって，いったいなに？　なんで全部ソフトにならなかったん？

小谷 実は大部分は音源です。松下はローランドとかと違って，自社でソフト音源持ってないんですよ。

かま やっぱり。だったら，音源ボード持っている人は，ソフトだけでええやん。

小谷 いえ，さすがにそれだけじゃありません。曲の長さを調節する機能のあたりも，ハードでやってるんです。

かま それって，ソフトでできへんの？

小谷 将来的にはできると思います。

画面4 ベースに奏でるメロディにはお馴染みの曲も多い

かま　ということはやな。将来，このハードがないソフト版が2万円ぐらいで発売されるんとちゃうん？

小谷　よくそんな恐ろしいこと，いいますなぁ。そんなこと雑誌に書かれたら，売れなくなってしまうじゃないですか。それに，そう簡単にはできませんよ。まず，AD/DA変換をパソコンの外で行うことで，ノイズを少なくするというメリットがあります。また，リアルタイムにBGMを生成する部分のソフト化は難しいでしょう。ソフト版を開発するとしても，機能を限定したものになると思います。

かま　それから，なんか音源がショボイような気がするけど。

小谷　そんなことは絶対にありません。いくつかの放送局で実際に音を聞いてもらいましたが，十分放送に使えるクオリティとの評価をいただいています。DoGAにあるスピーカーが悪いんとちゃいますか。

かま　それはいえる。次に，ソフトのほうの疑問点やけど，これって，自分で作った曲を登録して，いろいろアレンジするって使い方はできへんの？

小谷　プリセット以外の曲を追加する機能は搭載していません。メロディは基本的にMIDIのデータなのですが，指定した長さに自動編曲する処理なんかの都合で，一般のMIDIとは異なる部分があるんですよ。専用にデータを作れば追加することはできるはずですが，現段階ではフォーマットや追加方法を公開していません。

かま　演奏時間を指定できるのは非常にありがたいけど，テンポを操作できないのが不満。

小谷　だって，演奏時間にあわせてテンポを自動計算するんですから。ユーザー側で任意の演奏時間と任意のテンポを指定されたら，小節の途中で終わってしまうじゃないですか。

かま　いやいや，テンポは任意の値にする必要はないやん。現在のテンポより，1段階アップする，ダウンするでええねん。すると内部的には，1小節単位とかで増減する操作をすれば，小節の途中で終わるなんてことは起こらないやろ。

小谷　確かにそうですね。次期商品の改善提案として，検討したいと思います。

かま　そしてなにより問題なのは，リズムを120並べられても，どれが作品のイメージに近いのかさっぱり見当がつかへんことや。

小谷　「運動会」などのイメージを表すキーワードで分類してはどうかなどのアイデアもありましたが，実用的にするためには膨大なキーワードに対応する必要があり，事実上不可能でした。音楽の世界でも，そういったイメージによる分類というのは学術的にまだまだ遅れているんです。なんかうまい方法ないですかねぇ。

かま　学術的に正しくなくても，実用性があればええねん。たとえば音楽をキーワードで分類するんじゃなくて，逆に映像のほうを分類したらどうやろうか。映像を作っている人たちにアンケートして，自分の作品をイメージする言葉はなにかを集めて，そのなかで多いものを10ぐらいに絞るねん。“スピード感”とか“恐怖感”とか。そして，それぞれの曲について，各イメージにあう度合いを点数化するやろ。そうすれば，“スピード感は9で，恐怖感は2ぐらいの曲を検索”って使い方ができる。

小谷　本当にできると思いますか？

かま　いや，難しいと思う（笑）。

まとめ

ところで，DoGAでは毎年秋にCGアニメコンテストの入選者を集めた「CG合宿」というイベントを行っている。2000年10月に浜名湖で行われた「CG合宿2000」でも，このBGM-Gを展示したが，CGアニメ作家たちの感心は非常に強く，マウスを奪いあって使っていた。なかには，現在制作中の次の作品でさっそく使いたいという人もいれば，単におもちゃとして十分面白いという人も。そしてアンケートを行ったところ，購入希望価格にはかなりのばらつきがあったが，ほぼ全員の参加者がほしいと答えていた。

以上のように，BGM-Gはそれなりに実用性のある商品であるといえる。特に曲としてクオリティが高い点は評価されるが，その半面，映像にあった曲を見つける手段に欠けるのが問題だろう。完成度からいえば，まだまだ改善の余地はある。しかしそれは，まったく新しいジャンルを開拓した商品なのでやむをえないと思う。これ1台で映像制作における作曲問題がすべてクリアになるものではないが，少なくとも任意の長さに自動編集してくれるという点は画期的だ。だから，自動作曲技術というよりは，自動時間調整技術と認識するのが正しいかもしれない。今後，この技術をベースにして，どんどん後継機種を発表してくれることを強く期待する。

さて，前々から少し気になっていたのだが，CGの専門学校などで卒業制作をするとき，平気で既存の音楽をBGMとして利用しているケースが多い。個人ベースで楽しむならいざ知らず，それをそのままコンテストなどに出品したり，いろんな機会で上映したりするのはいかがなものだろう。CGのプロを養成する場で，著作権に対して杜撰な指導をするのは問題があるように思う。ということで，CG学校の皆さんはこのBGM-Gを導入されてはどうだろうか？

■注意

【著作権について】

BGMジェネレータで生成した曲は，映像作品やゲームなどのBGMとして，自主制作，商業ベースにかかわらず，自由に利用することができる。もちろん，CGなどのコンテスト用の作品に利用してもOK。

カタログにも「著作権フリー」と書かれているが，既存のメロディやリズムを組み合わせている以上，著作権法上厳密にいえば，曲の著作権は松下電器側にあるといえる。つまり，著作権フリーというよりは，著作権料フリーであり，上記のような使い方をする場合は著作権料を請求されることはないだけと考えたほうが正しい。

例外としては，たとえばこのBGMジェネレータで片っ端から曲を生成し，それを「BGM集」として販売するといったように，映像などを抜きにして，曲だけを商品化する行為は認められていない。

【サンプル曲】

本誌CD-ROMに，BGM-Gで生成したサンプル曲を収録している。曲の再生には，通常の音楽CDと同様にCDプレイヤーで聞くか再生ソフトでそのままOKだ。

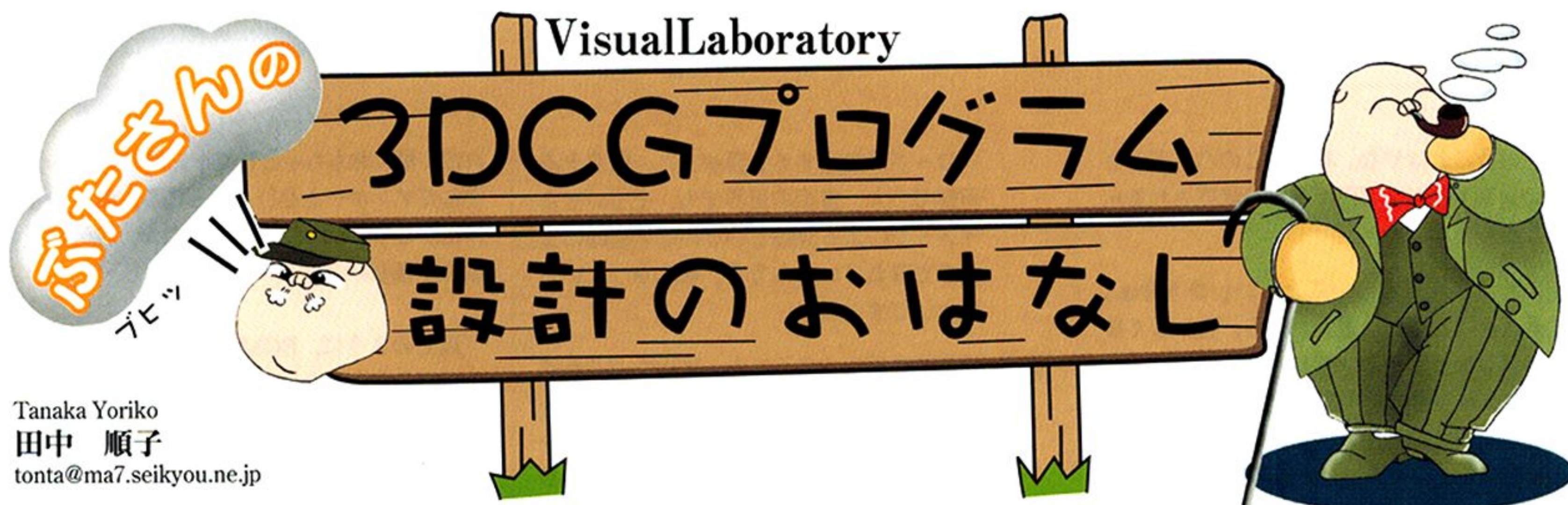
なお，このように，映像を伴わず曲だけ配布するのは，上記の［著作権について］の最後に触れたように，著作権上問題があるが，今回は松下電器様のご厚意により，特別に許可をいただいた。

■主な仕様

品番	：SY-VM1
対応OS	：Windows98
インタフェイス	：USB
入出力	：PHONES，LINE OUT，AUX IN，MIDI OUT，MIDI IN
出力ファイル	：WAV（44.1kHz，16bit ステレオ）
標準定価	：79,800円

http://www.panasonic.co.jp/technics/mi/vm1/

Visual Laboratory

ふたさんの3DCGプログラム設計のおはなし

Tanaka Yoriko
田中　順子
tonta@ma7.seikyou.ne.jp

Shade Plugin 入門

最近爆発的に人気の出ている国産のCGソフトウェアShadeを使って，独自のレンダラの開発に関して説明します。CG制作の立場から考えて，CGの表現力を向上させたいと思っている人の最初の一歩に役立つことができれば，と考えています。

はじめに

3D CGを描くにはなんらかの3D CG作成ソフトが不可欠です。2D CGでも状況は同じなのでしょうけど，ユーザーからの直接的な操作なしで計算によって画像を生成する3D CGの場合，使っているツールの機能/性能が作品により大きな影響力を持ってくるのは確かでしょう。

最近の3D CGツールはかなり多機能ですし，それを動かすマシンもどんどん高速になっています。それでも，3D CGを描いていて，自分が使っているソフトに「こういった機能があれば」とか「この部分を自動化してくれれば便利なのに」というような不満が出てくることがあります。あるいは，自分の思っている表現のために，計算アルゴリズムを改良するなど変更を加えたい場合もあると思います。

こういったことを実現するために多くのCGソフトウェアにはプラグインの開発環境が用意されています。ただ，ひと口にプラグイン開発環境といっても，対応状況はさまざまで，ソフトによっては思ったほど自由に機能の改良や変更ができないものも多く存在します。さらにプラグイン開発に関する情報の入手が困難であるために，思ったようなプラグイン開発ができない場合もあります。プラグイン開発者にとって，開発環境が整っているソフトウェアを探すことは重要なことだといえるでしょう。

そういったなかで，今回はプラグイン開発を行うプラットホームとしてShadeというソフトを取り上げてみました。

プラグインのなかでも，今回は画像生成部分つまりレンダラをプラグインとして追加することに焦点を当てて，その開発の基本的な部分に関して紹介していきたいと思います。

Shadeでプラグイン開発

さて，今回取り上げるShadeというソフトも，Professional版では，プラグインの開発環境が使えるようになりました。ご存じの方も多いでしょうが，このShadeは，現在，国産3D CGソフトのなかではトップの人気を持つものです。書店でShade関連の書籍がたくさん発売されているのを見かけた人も少なくないのではないかと思います。ただプラグインの開発環境に目を向けてみますと，Shadeはプラグインの機能ができてから日が浅いためか，開発環境がまだまだ満足できるものであるとはいえないかもしれません。

しかし，Shadeプラグインの開発環境は急速に整ってきています。Shadeの開発販売元であるエクスツールスが中心となってSDN（Shade Developer Network）という，Shadeユーザー，エクスツールス，サードパーティデベロッパとをつなぐネットワークが作られています。これに参加すれば，エクスツールスも含めたさまざまなデベロッパ間で情報の交換ができます。ここでいうサードパーティデベロッパは，一般的なソフトハウスのみを指すものではなく，有志で集まってシェアウェアやフリーウェアを開発しているアマチュアの開発者を含みます。Shadeのプラグインデベロッパには根っからのShadeファンが多く，たいへんな熱意を持って開発が行われています。プラグイン開発の情報は，これからどんどん充実していくことでしょう。

SDNへの参加

ということで，Shadeのプラグイン開発を行うのであれば，SDNに参加するべきでしょう。SDNに参加した場合のメリットとして，

1. プラグインIDを割り当ててもらえる
2. プラグイン開発に関して技術的な問題点等を話しあえる
3. 自分の開発したプラグインのフィードバックがすぐに返ってくる

ということが挙げられます。

まず，Shadeのプラグインは，それぞれ固有のIDが必要で，このIDのことをプラグインIDと呼びます。このIDは，ほかのプラグインと重複できないため，エクスツールスから割り当てられたものを使うのが確実です。

次に，技術的な問題点を話しあえるという点を考えてみると，SDN専用のメーリングリストがあり，ここで活発な討論が行われています。ここでは，エクスツールスの技術者やサードパーティデベロッパが多数参加して，それぞれの知識の交換が行われています。さらに，新しく開発したプラグインに関して，このメーリングリストで発表することにより，素早いユーザーの反応を受け取れるかもしれません。

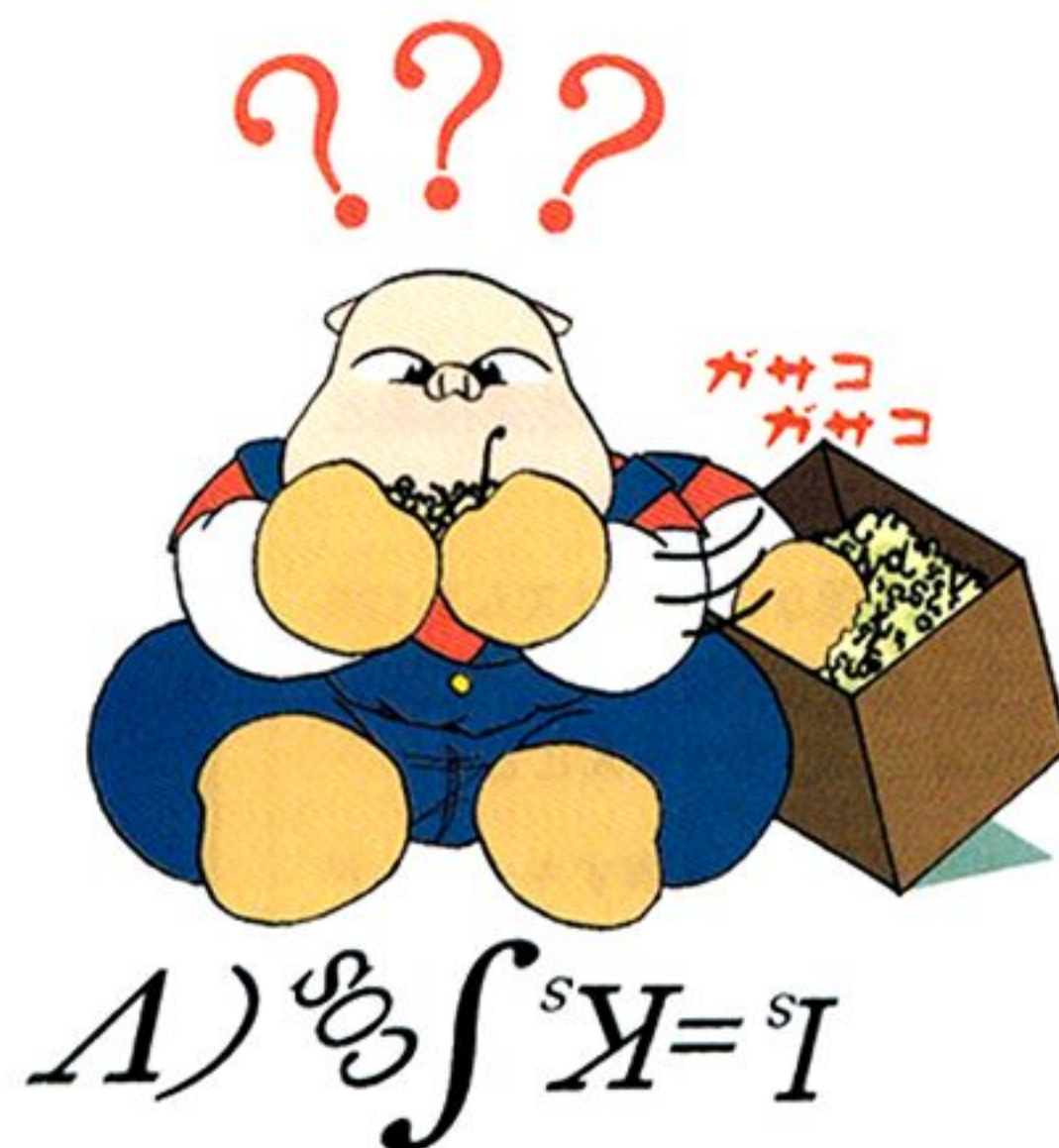



表　筆者の開発環境

OS	FreeBSD 4.0R
ビデオ	Matrox Millennium G400
CPU	Intel PentiumⅡ 350MHz×2
マザーボード	Giga-byte GA686BXD
メモリ	384MB

プラグイン開発を行う前準備

この原稿の執筆時点でプラグイン環境に対応しているShadeのバージョンは，R4のProfessional版であり，これを購入する必要があります。これには，Windows版（95/98/NT/2000），Mac版（MacOS8.1以上）があります。さらに，まだPreviewKitの段階ですが，Linux版（RedHat 6.0以降）のShadeも発売されており，Linux上でもShadeを用いてCG制作や，プラグイン開発が可能です。この記事ではLinux版のPreviewKitを用いて説明していきます。

Shadeのプラグイン開発は，Shade本体に付属している Shade Plugin SDK（以下SDK）を用いて開発を行います。このSDKは，随時リビジョンアップされており，エクスツールスのホームページから最新版をダウンロードできます。ここで，少し気をつけなければいけないのは，Shade本体もSDKもリビジョンアップが頻繁に行われているためプラットホームやリビジョンが異なると，微妙に仕様や動作が異なったりする場合があることです。

次に，開発環境として，Windows版は，Microsoft VisualC++ 6.0以降，Mac版はMetroworks CodeWarrior5.3以降が必要です。そして，Linux版は，ディストリビューションとしてRedHat 6.0以降に対応しており，これにインストールされている開発環境が必要となります。

筆者のShadeプラグイン開発は，Shade for Linuxをベースに行っています。ただし，筆者は根っからのBSDな人間なので，開発作業やプラグインの実行はFreeBSD上で行っています。FreeBSD上には，非常によくできたLinuxの互換環境があり，この上でLinuxのアプリケーションを動作させることができます。現在，筆者が使用しているFreeBSDのバージョンは，4.0RELEASEで，これには，RedHat 6.1ベースの互換環境がインストールされています。今回のShade for Linuxも，この機能を利用してFreeBSD上で動作させています。

Shadeプラグイン開発概要

Shadeには，プラグイン開発に関して，詳細なマニュアルが付属しています。そのため，インストールなどの詳細はそのマニュアルに譲って，ここではプラグインを実装する上での要点だけを述べます。

SDKを用いて作成されたプラグインは，ダイナミックリンクできる形のライブラリ形式で作成されます。これを作成するため，Windows版にはプロジェクトファイル，Linux版にはMakefileがそれぞれSDKに含まれています。これを使って，プラグインをビルドします。そしてできあがったライブラリ形式のファイルを，所定のディレクトリにコピーすれば，プラグインとして使用することが可能になります。Windows版であれば，拡張子が".dll"，Linux版であれば，拡張子が".so"というファイルができるはずです。Windows版は，Shadeがインストールされているフォルダの"Plugins"フォルダにこのDLLを，Linux版であれば，"/usr/local/shade/plugins/"に*.soをコピーすることによりプラグインとして認識されます。

Shadeのプラグインは，その機能によっていくつかのグループに分かれています。開発する目的によってどれを使うかを選択します。

(1) インポートプラグイン

ほかのソフトで作成した形状データなどを解析してShadeに取り込みます。一般的な目的としては，ほかのソフトとのデータコンバートに使用します。

(2) エクスポートプラグイン

Shadeで作成した形状データなどをほかのソフトで使用できるように内部のデータを変換し出力します。インポートプラグインと同様に，ほかのソフトとのデータコンバートに使用することが多いでしょう。

(3) クリエイトプラグイン

形状を作成するためのプラグインで，独自の形状を作れます。この手のプラグインは，人間の手で作成するのはたいへんな労力が必要でもコンピュータに計算させれば比較的簡単に作れるような形状を生成するときに用います。たとえば，フラクタルを用いて海岸線や樹木などを生成するのに有効です。

(4) モディファイプラグイン

形状の変換や編集を行います。これは，単純な形を入力して，それをより複雑な形状に変化させていくような作成支援を行うのに便利な機能です。

(5) エフェクトプラグイン

レンダリングの前処理や後処理で，エフェクトを与えます。たとえば，画像をぼやかしたり，光の効果を与えたりするようなときに使用します。

(6) アトリビュートプラグイン

形状，カメラ，無限遠光源などに，プラグインで使用する独自の属性を持たせることができます。

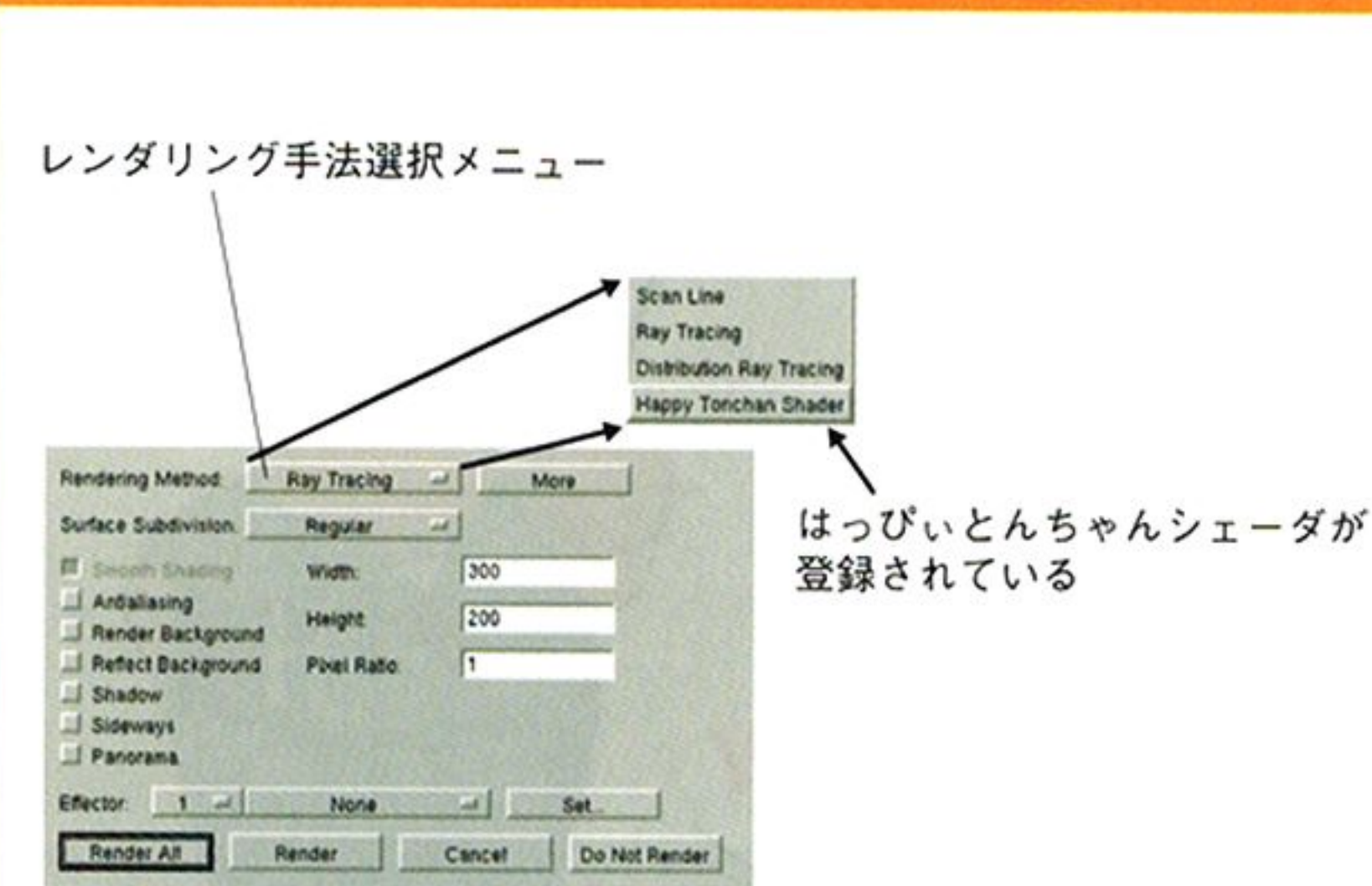
ちょっと特殊な例ですが，たとえば，物理的なシミュレーションを行ってそれを映像化したようなアニメーションを作成する場合を考えてみます。このとき，シミュレーションエンジンに与えるパラメータをアトリビュートに与えておくことができれば，データ構造を統一的に扱うことができるので，データ管理が楽になります。

(7) レンダラプラグイン

独自のレンダリングエンジンを追加することができます。

さらに，上記のプラグインを複数組み合わせて使用するマルチプラグインがあります。Shadeで開発できるプラグインは，以上のどれかに属することになります。

Shadeのプラグインは，C++で開発します。そして上記のように分類されたプラグインの各機能は，それぞれクラスで記述されています。そのクラスをオーバーライドすることにより，Shadeにプラグインとして新機能を追加します。SDKには各機能ごとにサンプルプログラムがありますので，最初は，これらを書き換えていくことがプラグイン開発の早道となるでしょう。ここでは，このプラグインに焦点を当てて説明を行っていきます。

図1　レンダラオプションにプラグインレンダラを登録

レンダラプラグインの構造

今回はオリジナルのレンダラを組み込むので，レンダラプラグインを追加します。レンダラプラグインは，shader_interfaceクラスから派生したrendering_interfaceクラスをオーバーライドすることにより実装します。今回のレンダラプラグインは実際には以下の3つの関数を書き換えるだけで，実現できています。

(1) get_id()

プラグイン ID を返します。プラグインIDとは，そのプラグインを識別するコードですので，プラグインごとにユニークでなければなりません。

```
// プラグインIDを返すextern_c
 int STDCALL get_id (const IID &iid, int i, void *) {
return tonchan_renderer_id;       // tonchan_renderer_
                                   idには実際には16進
                                   数のIDが入っている}
```

(2) get_name()

プラグイン名を返します。プラグインに開発者がつけた名前が登録されます。この名前は，Shadeのメニューなどに登録されて，ユーザーが選択できるようになります。この名前がレンダラを起動したときのレンダラオプションに登録され，新しいプラグインレンダラが選択可能となります。

```
// プラグイン名を返す
extern_c const char * STDCALL get_name (const IID
&iid, int i, shade_interface * shade, void *) {    return
"Happy Tonchan Shader";}
```

(3) render()

レンダラ本体を記述します。この部分にオリジナルのレンダリングアルゴリズムを記述して，画像の生成を行います。

レンダリングとは

本稿では，特にレンダラプラグインの開発を目的とします。そこで，ここではレンダラプラグインを開発するうえでの基礎的な理論の話を行います。レンダリングとは，最終的には画像上の各ピクセルの色を決定することです。一般的に，3D CGというのは，物体が立体的に見えるように色づけされたCGのことを指します。

太陽や電球などの光源から出た光が，物体上で反射されて目に届き，それが脳で解釈されてものを見ることができます。一般的な光源から出た光は，多数の波長の光により構成されています。光が，物体上で反射するとき，物体表面の性質により特定の波長のみを反射します。そして観測者の目には，到達した光に含まれている波長により色が認識されます。これをコンピュータで表現できるような形で表現します。

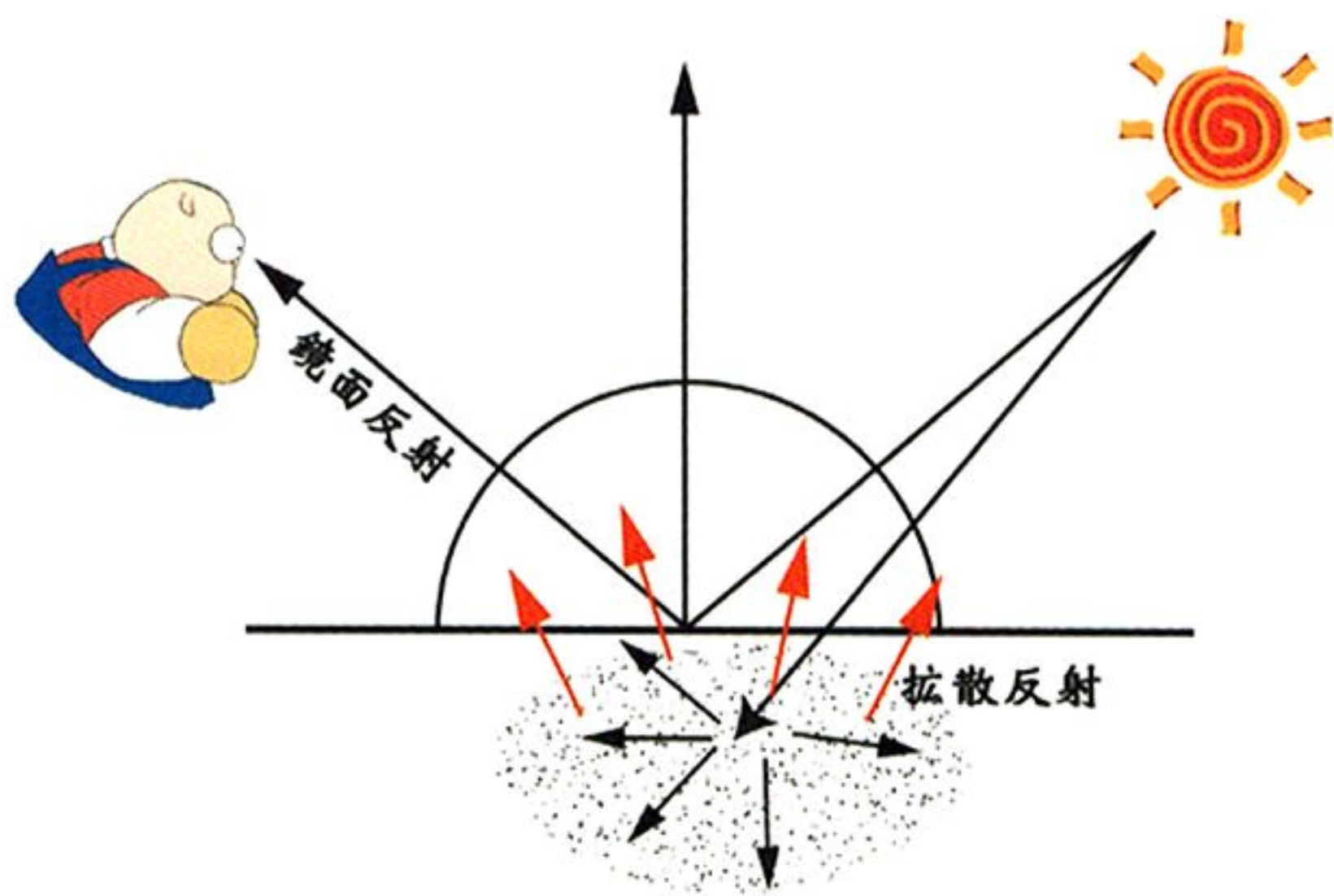
図2 Rendering Equation の概念図

一般的に現実の複雑な現象等を単純化して表すことをモデル化するといいますが，ここでも光源と観測者の幾何学的な関係と物体の材質などをモデル化して，それをコンピュータで計算させることにより物体の色を計算し映像化します。

The Rendering Equation

まず，映像化のためのモデルを紹介します。ここではいきなりレンダリングの最終兵器と呼べるものを少しご紹介します。照明モデルに関する研究のひとつで，1986年にKajiyaの"The Rendering Equation"という論文が発表されています。ここでのKajiyaの功績は，レンダリングに関する理論を統一し，ひとつの式にまとめあげたことでしょう。そのため，ここで提案されている方程式を解けば，この世の中にあるすべての物体の任意の座標に対しての輝度の決定ができます。さらに，これを光の波長に関して計算すれば，すべての座標上で色が決定できる「究極のレンダラ」ができるというものです。

しかし，この方程式は解析的に解くことができないため，通常は近似解を求めることになります。この理論では，リアルなレンダラを開発するということは，いかに厳密にこの方程式を解くかということに集約されます。一般的に知られているレイトレーシング法やラジオシティ法などのレンダリング手法は，ある意味，この方程式を近似的に解いているようなものだといえるでしょう。この Rendering Equation は，任意の座標 x での輝度の計算を次式のようにまとめています。これをすべての座標上で計算すれば，す

図3　Lambertの余弦則の概念図

べての座標での輝度が求まります。

$$I(x,x')=g(x,x')\left[\varepsilon(x,x')+\int_s \rho(x,x',x'')I(x',x'')dx''\right] \quad (1)$$

は，それぞれ位置パラメータであって，それぞれ座標を示します。ここでは局所的に見れば，座標からの光が 座標x'で反射して，座標xに届いている様子をイメージしていただければ，この式の意味はわかりやすいと思います。この式の各項を説明しますと，

- $I(x,x')$　：x' から xへの輝度
- $g(x,x')$　：xとx'の幾何的条件を与える項
- $\varepsilon(x,x')$　：x' が自ら発光している場合，xへの輝度（x'で発光してxに届く）
- $\rho(x,x',x'')$：& x''から発した光が x'を経てxへ到達した輝度（x'での反射率に相当する）

のようになります。ただし，x'に入射するエネルギーは，この座標に影響を与える全方位からの総和のx''を微小立体角に関して積分します。これを図で表したものが図2となります。x'は，x''から到達した光を反射するか，あるいは自らも発光する場合があります。そしてその光をxで観測した場合の概念を幾何学的に表しています。

レンダラの設計

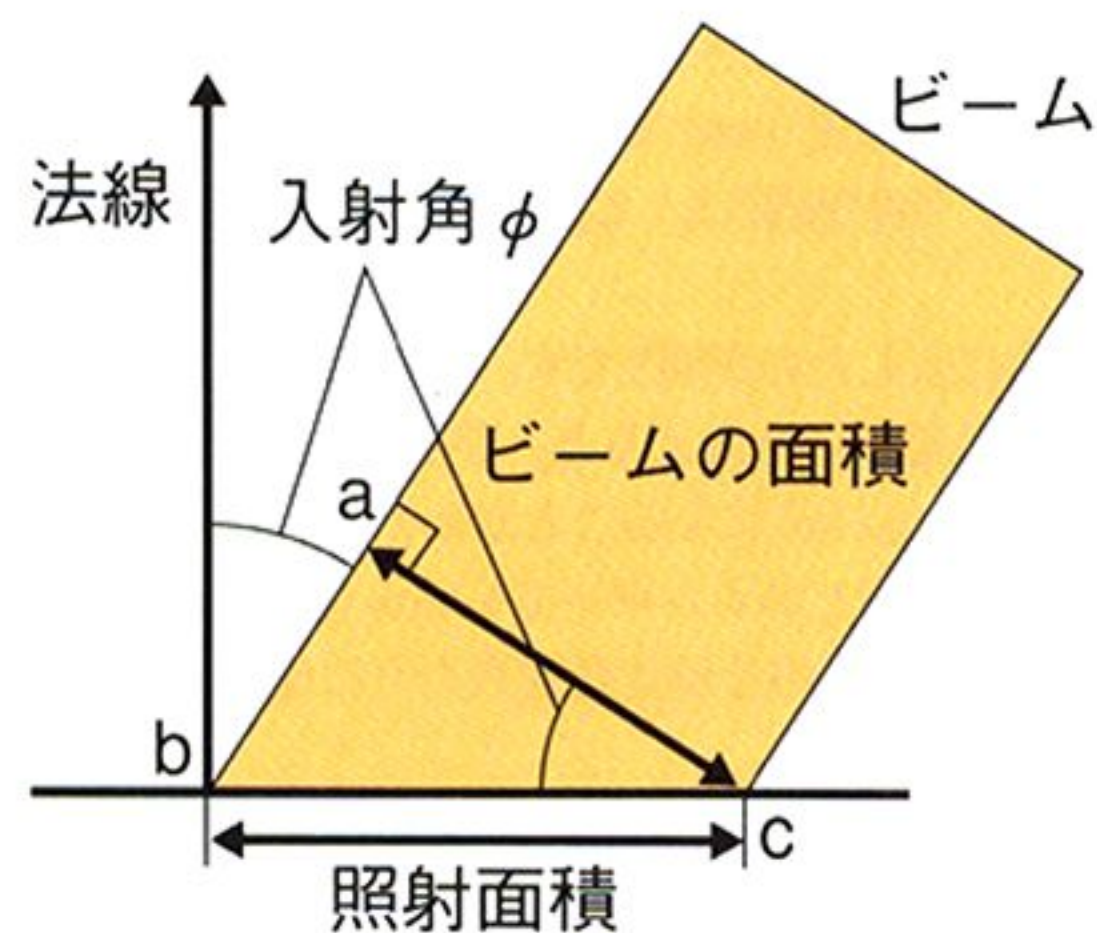
本稿のレンダラの設計方針は，Kajiyaの Rendering Equation を基礎とします。理想的には，Kajiyaの Rendering Equation を厳密に解けばよいのですが，現実にはそうううまくはいきません。そのため，影響の少ない部分を省略したり表現したい物体の性質に適したアルゴリズムを開発したりすることが必要です。レンダラの設計は，開発者の力量がもっとも問われる部分です。ここでは，もっとも単純な輝度計算モデルを例に挙げてその設計手順を示していきます。

まず，一般的な物体の表面で起こっている光の反射は，拡散反射，鏡面反射に分けることができます。拡散反射は物体表面に到達した光は物体表面から入り込んで内部で乱反射します。このとき色素により特定の波長が吸収され物体の色となります。鏡面反射は，物体内部に入り込まず物体表面で反射します。そのため物体表面で観測できる放射輝度は，次式のように示すことができます。

$$I(\lambda)=k_d I_d(\lambda)+k_s I_s(\lambda) \quad (2)$$

ここでは物体上の色を決めるために，輝度は波長lの関数として記述しています。k_dとk_sはそれぞれ拡散反射と鏡面反射の割合で。一般的には，

$k_d+k_s=1$

とします。

ところでlは，光の波長ですが，実際にCGで計算するときは，RGBの3つの色で計算するため，(3) 式のようにRGBそれぞれで輝度計算をすることにより求めることができます。今後の式の説明では，適時RGBの記述は省略する場合がありますが，輝度計算はRGBそれぞれに行い色を決定します。

$$I(R,G,B)=k_d I_d(R,G,B)+k_s I_s(R,G,B) \quad (3)$$

次に，簡単化のために微小な点を仮定し，座標x'付近の微小な領域だけで光の反射が起きていると仮定します。この場合，微小領域ですのでほかの物体に遮られたり，距離によって光が減衰したりすることがないと仮定できます。

こうすると，光の反射に影響するものは入射輝度と，面法線，光源方向，視線方向の3つのベクトルで表すことができます。このときx'での入射輝度をI_lとし，面法線をN，光源方向を向くベクトルをL，そして観測点を向くベクトルをVとします。ただし，各ベクトルは大きさが1になるように正規化して，方向のみを持つようにします。

拡散反射の設計

拡散反射の様子は，左上の図のように物体表面に到達した光が物体内部で乱反射します。理想的に乱反射した光は，どの方向にも均一に進みます。そのため，観測できる光の強さは観測位置には依存しなくなります。

拡散反射は，物体表面がどの色を反射するのか，反射点にどの程度の強さの光が入射するのか，全体の反射光の中での拡散反射成分の割合などにより決まります。入射光に関しては，入射角度が小さいと，単位面積当たりのエネルギーが大きくなり，逆に入射角度が大きくなると，単位面積当たりに受けるエネルギーが小さくなるという性質があります。これは，夏は太陽光の入射角度が小さいため暑くなり，冬は入射角度が大きくなるため

に寒くなるのと同じ理由です。

これを，Lambertの余弦則に基づいて説明します。微小立体角の光束を考えると平行ビームを仮定できます。入射角度と単位面積に照射されるエネルギーは，入射角の余弦（cos）で求めることがわかります。入射光が斜めから入射した場合，照射面積は，ビームの幅よりも太くなることがわかります。

これを二次元的に表したのが図3です。この場合，照射部分とビームの幅で作られる直角三角形abcを考えると，照射面積に当たるbcとビームの幅に当たるabの比は，cos (φ) になることがわかります。つまり，ビームが面法線に対して傾けば，同じエネルギーで1/cos (φ) 倍の面積を照射しなければなりません。これは，その分単位面積当たりに受ける光のエネルギーが減ることを意味します。

このため，反射率は$\rho(x,x',x'')$=cos (φ) となります。ここでφは入射角，k_dを全体の反射光内の拡散反射光の割合とし，$C_{R,G,B}$は，それぞれRGBの反射率とします。そうすると，拡散反射をRendering Equation風に書けば(4)式のようになります。

$$I_{d(R,G,B)}(x,x') = k_d \int \cos(\varphi(x'))\, C_{R,G,B} I(x',x'')dx'' \quad (4)$$

そして，この式を先ほど定義したベクトルで記述すると(5)式のように書き直すことができます。

$$I_{d(R,G,B)}(x,x') = k_d \int (\mathrm{NgL}(x''))\, C_{R,G,B} I_l dx'' \quad (5)$$

鏡面反射の設計

鏡面反射は，物体内部に入り込まずに物体表面で反射した反射光です。まず，理想的な鏡の鏡面反射を考えます。xで観測される輝度は(6)式のように表されます。理想的な鏡では，$\rho(x,x')$が入射光と反射光が，x'において正反射の関係のときのみ$\rho(x,x',x'')$=1となり，それ以外は$\rho(x,x',x'')$=0となります。

$$I_s = (x,x') = K_s \int \rho(x,x',x'') I(x',x'')\, dx'' \quad (6)$$

しかし，実際の物体は完全鏡面でない場合も多く，表面がざらついているような場合があります。このような場合は，先ほどのように関数ρの値がパルス的に変化するのではなく，もっと緩やかに変化します。これは，ザラツキの凹凸によって生じた微小面の法線方向がばらつくためです。

鏡面反射の反射率は，この分布を表す関数により決まります。この部分は，複雑な反射特性を持つため，多くの場合，簡略化したモデルで近似します。ザラツキのないツルっとした表面は，鏡に近くなるので，この反射の特性がシャープで輝度が急激に変化します。逆にざらついた面は光が散乱されるのでソフトな輝度変化の特性を持ちます。

この反射特性を記述する方法のひとつとして，有名なものにPhongモデルがあります。このモデルは入射光と反射光が正反射の関係に近いものほど強く反射し，逆に正反射の関係から離れるほど反射が弱くなると仮定しました。そしてこの特性をPhongは，$\cos^c(\alpha)$という関数で近似しました。

本稿でも，Phongモデルを用いて鏡面反射部分を設計しました。そうすると入射光の正反射方向と光源方向ベクトルのなす角をαとすると鏡面反射は次式で表すことができます。ここでcの値が大きいと，分布の狭いシャープな関数となり，cの値が小さいと分布の広いなだらかな関数となります。これは，cを調整することにより物体のザラツキ感を調整できることを示します。

$$I_s = K_s \int \cos^c(\alpha) I(x',x'')dx'' \quad (7)$$

そしてこれも同様に，ベクトルにより記述すると(8)式のようになります。ただし，V_rは，VのNに対する正反射ベクトルです。

$$I_s = K_s \int (V_r \cdot L(x''))^c I_l dx'' \quad (8)$$

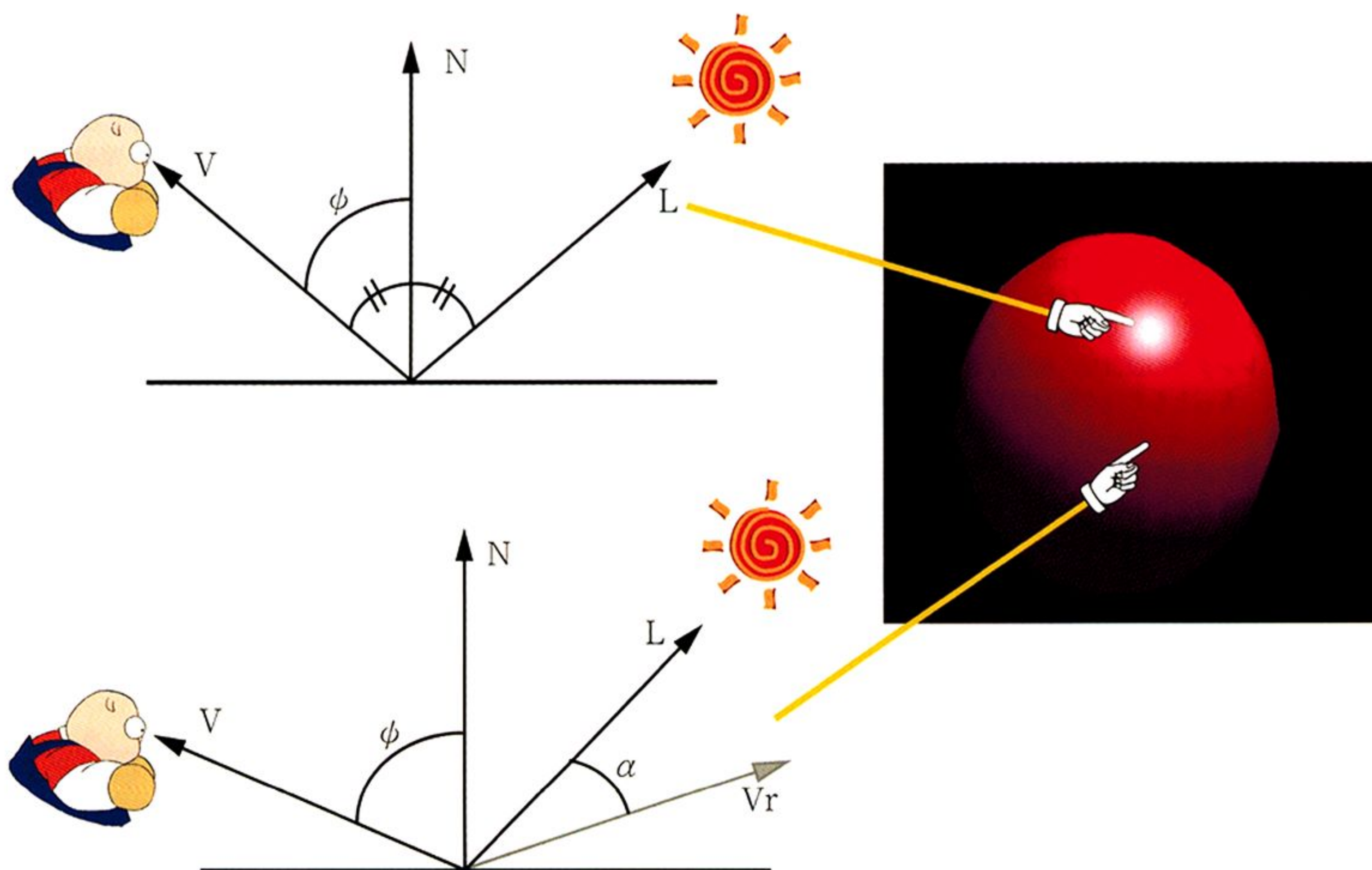
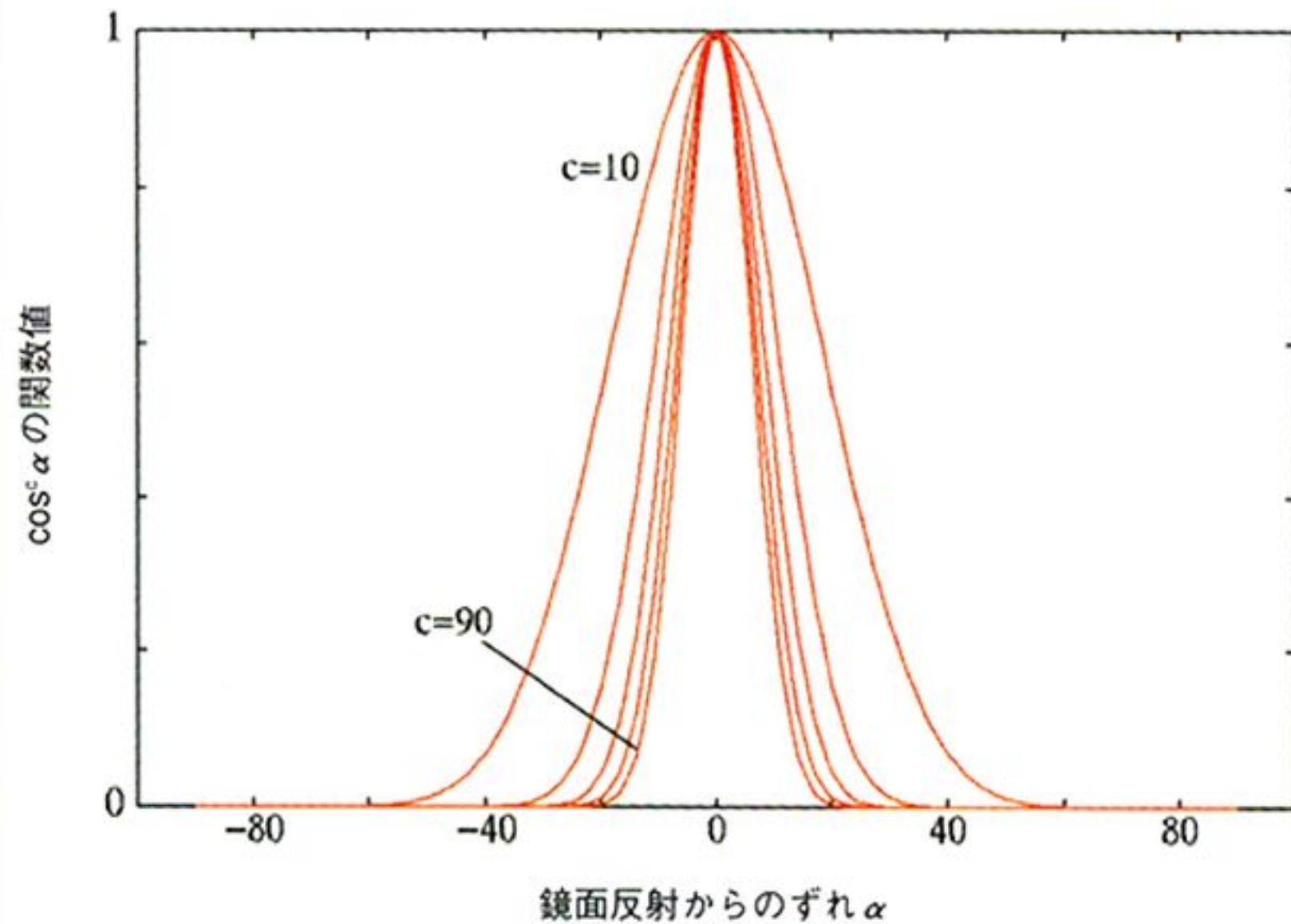



図4 $\cos^c(\alpha)$のグラフ

ここでひとつ気づかれたと思いますが，鏡面反射には色を示す部分がありません。実は，一般的な物質の場合は，鏡面反射は，光源の色をそのまま反射します。そのために鏡面反射は，光源の色がそのまま観測点に届きます。つまり，物体上でキラッと光っている部分というのは，光源の色がそのまま見えているわけです。

金属の反射の設定

いままで設計を行ってきたモデルは，一般的な物体に対してある程度有効です。しかし，これらがすべての物体に対して適用できるかというとそううまくはいきません。たとえば，金属を表現しようとした場合上記のモデルではうまく表現できないことがわかります。

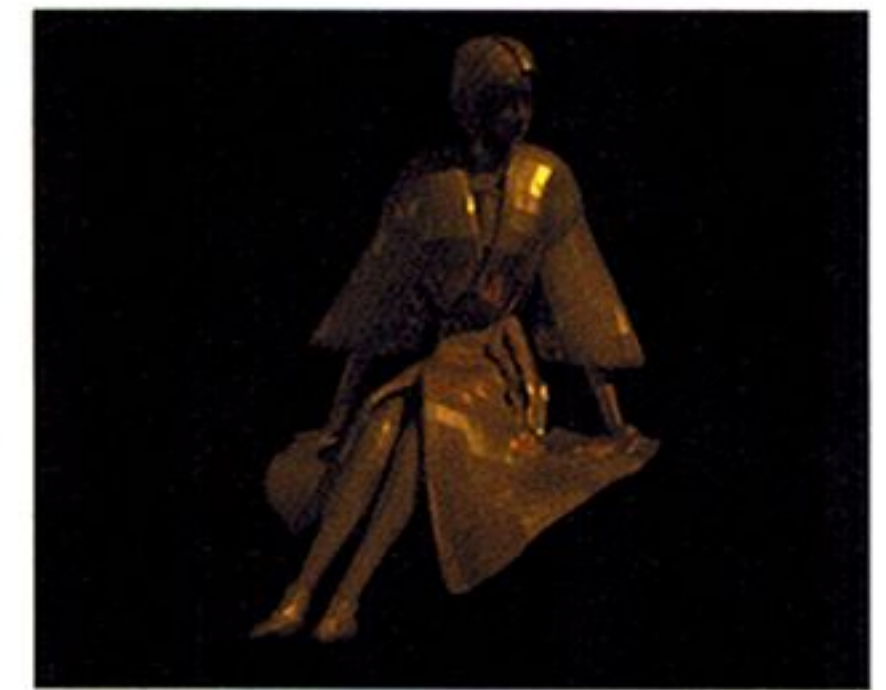
さて，そうなると金属らしさというのはどこから出てくるのでしょうか。金属の色を計測器で計測し，そのRGBの値を得たとします。そしてその色をなにかに塗って見た場合，どのように見えるでしょうか。その塗ったものはまず金属には見えないでしょう。

実は，金属とプラスチックなどの物体とでは反射のメカニズムが異なるのです。そのため，通常のモデルではうまく金属の質感を表現することができません。具体的にどのように違うかといえば，まず金属では拡散反射が存在せず，鏡面反射のみしか起こりません。しかも，通常の物体と違って金属では光の波長によって反射率が異なるため，鏡面反射で色が決まります。つまり，いわゆるキラッと光る部分の色が光源の色ではなくて，その金属独特の色になるということです。この現象は幾何学的な条件によって見えが異なります。このため幾何学的な反射率の変化の影響を考慮せずに単純に色を取り出しただけでは，金属には見えないのです。

さて実際に金属反射を設計します。本来は，金属に適した反射のモデルというのがあるのですが，今回は，もっとも単純なモデルを用いて設計を行います。先ほど設計した鏡面反射の部分を改良して金属らしさを表現してみます。金属は，拡散反射が存在せずに，鏡面反射だけで色が決定します。そのため，金属は鏡面反射で色が決定するために(9)式のように記述します。

$$I = \int \left(\mathrm{V}_{\mathrm{r}} \cdot L(x'')\right)^{c} C_{\mathrm{R,G,B}} I_{\mathrm{l}} dx'' \quad (9)$$

さてここで，同じ色でもこのような反射モデルに当てはめた場合と，単純にベタ塗りした場合の違いの例を示します。これは，理科年表から金(gold)の分光反射率に人間の目の波長ごとの感度を掛けて，さらにRGB値に変換したものを用いました。この2つの画像を比較してみると，同じRGB値でもまったく質感が違うことがわかります。このように金属の質感を出すためには，なんらかの形で金属反射の特性を与える必要があります。

図5 下記の2つの図は，ともに同じRGB値で色づけしています。

金色(ベタ塗り)　金色(金属反射の特性を与えたもの)

レンダラプラグインの実装

これまでレンダラの設計を行い，そのモデルの構築を行ってきました。ここからは，この設計したモデルを用いて，実際にレンダラのプログラムを作成し，プラグインに実装します。

拡散反射の実装

まず，(4)式は，積分を含むのでそのままではプログラムにできません。そこで(10)式のように離散的に記述します。ただし，ここでは立体角の大きさは一定として無視して記述しています。nは，入射光を有限個のビームに分割したと仮定したときのビームの本数とします。

$$I_{\mathrm{d}} = K_{\mathrm{d}} \sum_{i}^{n} \left(N \cdot L_i\right) C_{\mathrm{R,G,B}} I_{\mathrm{l}} \quad (10)$$

これをプログラムで記述します。ただし，次のプログラムではアルゴリズムだけを示しています。具体的な値を変数に代入していないので，このままでは実行できません。

```
vec3  id;     // 拡散反射の強さ
float il;     // 入射光の強さ
vec3  bc;     // 反射率(RGB)
vec3  nv;     // 法線(ベクトル型)
array_<vec3> lv = new vec3[n]; // 入射光の方向(反射点
                                  にn個の光が入射する)

// RGB各色の初期化
id[0] = id[1] = id[2] = 0.0;

// すべての入射光に対して拡散反射の強さの計算
for( int i = 0; i < n; i++) {
    for( int j = 0; j < 3; j++) { // RGB それぞれについて計算
        float p;
        // N と L の内積計算
        p = nv.x * lv[i].x + nv.y * lv[i].y + nv.z * lv[i].z;
        id[j] += p * bc[j] * il; // 各色の拡散反射の計算
    }
}
```

鏡面反射の実装

これも拡散反射の実装部分と同様に離散的に記述しました。その式は次式のようになります。

$$I_s = K_s \sum_i^n \left(V_r \cdot L_i\right)^c I_1 \quad (11)$$

これをプログラムで記述します。ただし，次のプログラムもアルゴリズムだけを示しています。

```
float is;      // 拡散反射の強さ
float il;      // 入射光の強さ
vec3 nv;       // 法線(ベクトル型)
vec3 vv;       // 視線方向(ベクトル型)
vec3 vvr;      // 視線方向の正反射方向(ベクトル型)
array_<vec3> lv = new vec3[n]; // 入射光の方向(n個のベ
                                  クトル配列)
float dnv;     // N と V の内積
float dnv2;    // N と V の内積の2倍の値
float c;       // 鏡面反射の鋭さを表すパラメータ

// 内積計算
dnv = n.x * v.x + n.y + v.y + n.z + v.z;
dnv2 = nv * 2.0;

// 正反射方向を求める
vvr.x = dnv2 * nv.x - vv.x;
vvr.y = dnv2 * nv.y - vv.y;
vvr.z = dnv2 * nv.z - vv.z;

is = 0.0;
for( int i = 0; i < n; i++) {
    float p;

    // L と Vr の内積計算
    p = (vvr.x * lv[i].x + vvr.y * lv[i].y + vvr.z * lv[i].z) * il;

    p = pow( p, c); // 鏡面反射の鋭さの計算
    is += p;
}
```

金属の反射の実装

金属の反射は，鏡面反射成分のみで色の計算を行っているのが特徴です。金属反射を離散的な式で表すと次式のようになります。

$$I = \sum_i^n \left(\mathrm{V_r} \cdot \mathrm{L}_i\right)^c C_{\mathrm{R.G.B}} I_1 \quad (12)$$

```
vec3 is;
    is[0] = is[1] = is[2] = 0.0;
    for( int i = 0; i < n; i++) {
        float p;

        // Vr と L の内積計算
        p = (vvr.x * lv[i].x + vvr.y * lv[i].y + vvr.z * lv[i].z) * il;

        p = pow( p, c); // 鏡面反射の鋭さの計算

        // 金属の色の計算
        for( int j = 0; j < n; j++) {
            is[j] += p  * bc[j];;
        }
    }
```

プラグインの作成例

これまで述べたことを基に実際にレンダラプラグインを作成した例を挙げます。このプラグインは，レンダラプラグインとしてShadeのレンダラに追加されます。プラグインの処理内容としては球を3次元的にレンダリングするだけの単純なものです。しかも，3次元的といっても今回は，極力簡略化するために透視変換（遠近感をつける）を行わずに単純な並行投影で，光源は右上から当てています。反射は，拡散反射と鏡面反射で構成されて，入射光束は1本だけと仮定して簡略化しています。実用的なプラグインとはいえませんが，一応，レンダラとしては最低限の機能を持ちます。

```
//
// はっぴぃとんちゃんレンダラ( tonchanrenderer.cpp)
// Copyright Studio Happy Tonchan.
//

    virtual void render (void * = 0) {
        point_class img_size;
        float        rad;

        // image_interface の取得
        compointer<image_interface> image =
            rendering_context->get_image_interface();

        // イメージサイズを取得
        img_size = image->get_size();

        // 半径の設定(画像の縦のサイズの半分)
        rad = float( img_size.v / 2.0);

        // 光源方向ベクトル(光源は右上から照射)
        vec3 lvec( -1.0, -1.0, 1.0);
        normalize( lvec);

        // 視線方向ベクトル(視線方向は Z 軸と一致)
        vec3 vvec( 0.0, 0.0, 1.0);
        normalize( vvec);

        // 球の色(赤)
        vec3 bcolor( 1.0, 0.0, 0.0);

        // 光源の色(白)
        vec3 lcolor( 1.0, 1.0, 1.0);
```

```
// 拡散反射の割合
float  kd = 0.5;

// 鏡面反射の割合
float  ks = 0.5;

// 鏡面反射の鋭さのパラメータ
// c を大きくするとツルっとした面
// c を小さくするとザラついた面
float  c  = 60.0;

for( int i = 0; i < img_size.v; i++) {
    for( int j = 0; j < img_size.h; j++) {
        const int ofst_x = 20, ofst_y = 0;  // 画面の
                                               オフセ
                                               ット
        rgb_class rgb; // 画素の値
        float    d;      // 拡散反射成分の強さ
        float    s;      // 鏡面反射成分の強さ
        vec3     nv;     // 法線
        float    x, y;   // ワールド座標の x, y
        float    t;      // テンポラリ変数

        // デバイス座標系からワールド座標系に変換
        // そんなに大げさなものじゃないけど……
        x = float( (i - ofst_x) - rad);
        y = float( -((j - ofst_y) - rad));

        // 画素位置に球が存在するかどうかの判定
        t = - (x * x) - (y * y) + (rad * rad);
        if( t >= 0) {

            // 法線計算
            nv.x = x;
            nv.y = y;
            nv.z =  float( sqrt( t));
            normalize( nv);

            // 拡散反射成分の強さ
            d=nv.x * lvec.x+nv.y * lvec.y+nv.z * lvec.z;

            if( d < 0.0) // 面の反対側
                d = 0.0;

            // 鏡面反射成分の強さ

            // 視線ベクトルの正反射方向の計算
            vec3  vvr;
            float dnv2, dnv;
            dnv =
                (vvr.x * lvec.x +
                 vvr.y * lvec.y +
                 vvr.z * lvec.z);
            dnv2 = dnv * 2;
            vvr.x = dnv2 * nv.x - vvec.x;
            vvr.y = dnv2 * nv.y - vvec.y;
            vvr.z = dnv2 * nv.z - vvec.z;
            normalize( vvr);

            // ハイライトの鋭さの計算
                (vvr.x * lvec.x +
                 vvr.y * lvec.y +
                 vvr.z * lvec.z);

            s = pow( p, c);

            // RGB それぞれに輝度計算
            for( int k = 0; k < 3; k++) {
                rgb[k] = float(
                    ( ( kd * (d * bcolor[k])) +
                      ( ks * s) ) * lcolor[k]);
            }

            // 画素に色をつける
            image->set_pixel( j, i, rgb);
        }
    }
  }
}
```

終わりに

今回は，まず，Shadeのプラグイン開発に関する環境の説明を行いました。次に，レンダリングに関する理論の基礎的な部分の概要を述べ，それを用いての設計技法を示しました。次にそのモデルをShadeのプラグインとして実装する方法を示しました。これで3D CGというのは，比較的単純で短いプログラムだけで簡単に描けるということがわかっていただけたと思います。

ただ，説明をわかりやすくするため，あるいは筆者の不勉強のせいで，数学的な厳密さが損なわれている部分がありますが，そこは温かい目で見てください。

次回は，実際にShade本体から形状データや光源データを取り出して，実用的なプラグインにすること，より高度な照明モデルを用いてCGのリアルさを向上させる方法も解説していきたいと思っています。そして応用としてレイトレーシングプラグインを作成して，そのなかで必要になる実装方法や高速化手法などを同時に解説していければと思っていますが，どこまで実現できるかは未定です。それではまた。

図6　本稿のレンダラプラグインで作成した球

Visual C++で始めるWindowsプログラミング（5）
レイヤードウィンドウってなんだ

菊地 功 Kikuchi Isawo

今回はWindows2000以降から搭載された新しい機能，レイヤードウィンドウを扱います。画面ににゅっと出てきたり，半透明でふわっと現れる新しいウィンドウの描画形態ですが，非矩形ウィンドウも自然に作れますし，ウィンドウ自体をキャラクターのように動かしたりといったことも自在です。加えてWindows98でのアルファ付きBLTについても紹介します。

コンシューマWindowsもWindows Meで最後。でもWindows 98とたいして変わらない（Windows 2000ともあまり変わらなさそう）ということで，そんならばいっそWindows 2000にしてしまおうかという人も少なくないはず。ドライバ環境もそこそこ整ってきているし。

で，2000を入れてみて，やっぱあんまり変わんないかも，と思うのだが，最初にスタートメニューを開くと，ちょっと驚くかもしれない。ぼわわっとフェードインするメニューに（設定でぺろぺろっとプルダウンするメニューにもなる）。所詮はイロモノ，うざったいと思うかもしれないが，そう結論を急ぐことなかれ。フェードイン，つまりは半透明，これはメニューだけではなく，通常のウィンドウにも実装できるAPIによるものなのだ。フェードイン／アウトだけならば，スプラッシュウィンドウくらいにしか使えないかもしれないが，もっと汎用的な半透明ウィンドウならば，いろいろと用途はありそうだ。この半透明なウィンドウを，レイヤードウィンドウという。

今回は，このレイヤードウィンドウで遊んでみよう。なお，レイヤードウィンドウはWindows 2000からの対応（多分Windows Meでも利用可能）なので，Windows 98以前のOSでは利用できない。あしからず。

ところで，レイヤードウィンドウ関連のAPIのライブラリは，Visual C++ 6.0（VisualStudio 6.0）やサービスパックには入っていない。これらのライブラリとヘッダは，http://www.microsoft.com/msdownload/platformsdk/setuplauncher.htmからダウンロードできるPlatform SDKに含まれている（原稿執筆時点の最新版はJuly2000）。

ただし，このSDKは500MBにも及び，ちょっとダイヤルアップでは落とせるサイズではない。それどころか，サイトが尋常ではないほど重いので，専用線でも一筋縄では落とせない。ひょっとしたらCD-ROMで配布をしているのかもしれないが，筆者が探した限りでは見つからなかった。できれば本誌に収録したかったのだが，どうもそれも無理っぽい。とはいえ，レイヤードウィンドウがやりたい程度なら，実は必ずしもPlatform SDKは必要なかったりする。そんなわけで，種明かしはあとに譲るが，ここではPlatform SDKがなくてもコンパイルできるようにサンプルを作ってあるので，ご安心めされい。

レイヤードウィンドウの仕組み

通常Windowsでは，ウィンドウへのGDIなどによる描画は直接VRAMへレンダリングされる。これを半透明に対応させようと思ったら，半透明のペンを用意することでは対処できない。なぜなら，重ね描きした部分はどんどん濃くなってしまうし，ウィンドウを移動して背景が動いてしまうと，逐一ペンで描き直さねばならなくなってしまうからだ。そのたびにメッセージを発し，ユーザーに再描画を促していたら，とてもではないが重くて動かない。ではどうするか。レイヤードウィンドウでは，バックバッファを取ることで対処している。ユーザーの描画はいったんバックバッファにレンダリングされ，それがディスプレイに描画される必要のあるときに，システムがVRAMに転送・合成する。これが「レイヤード」たる所以である（と思う）。ワンクッション必要になるが，この方法ならばハードウェアを利用しやすいので，アルファに対応したビデオカードがあれば容易に対応できるというわけだ。

さてこのレイヤードウィンドウだが，実際にコーディングするには2つの方法がある。高水準な方法と低水準な方法と考えてもらって差し支えないだろう。いってみれば，簡単だが融通が利かない方法と，ちょっと面倒だが応用の利く方法だ。サンプルを交えながら，順番に説明していこう。

SetLayeredWindowAttributesによる高水準な手法

こちらの方法は，とにかく簡単である。前述のバックバッファの取り扱いも全自動で，ユーザーはレイヤードウィンドウをまったく意識する必要がな

COLUMN 1　WindowsMeはレイヤード非対応

記事中ではレイヤードウィンドウはWindows Meでも多分可能と書いたが，その後にWindows Meをいじってみたところ，メニューのフェードインオプションがないし，マウスカーソルに影も落ちていない。変だなぁ。どこぞでβを見たときには影が落ちていたような気がしたんだが。これはレイヤードウィンドウ対応していないのでは？　と思い，今回のサンプルのlayered3を実行してみたところ，バージョンチェックで引っかかってしまう。さらにおや？　と思い，システムのプロパティを見ると4.90.3000。そうかぁ…5.0じゃなかったんだ。試しにバージョンチェックを外してみたら、案の定SetLayeredWindow-Attributesやらが実装されていない。そうするとコンシューマWindowsのレイヤードウィンドウ対応は，NTとカーネルが一緒になってからか。そんなに難しいことか？そんなわけで，みんな2000を導入するのだ（といいつつ，うちでは2000を降ろしてしまった。またどこかに入れなきゃなぁ）。

い。従来のウィンドウと違う部分は，たった2つだけ。ひとつは，ウィンドウの作成の際に，拡張スタイルとしてWS_EX_LAYEREDを指定すること。具体的には，CreateWindowEx()の第1引数でそのフラグを指定すればよい。これにより，システムはこのウィンドウがレイヤードウィンドウとして扱われることを認識する（低水準な手法もこれは同じ)。もうひとつは，ウィンドウが作成されたあとに，SetLayeredWindowAttributes()でレイヤードウィンドウの属性を設定すること（コラム1)。このAPIにより，システムは自動的にバックバッファを作成し，ユーザーに渡すウィンドウのデバイスコンテキストを，このバックバッファのデバイスコンテキストに差し替える。これだけだ（図1)。

たとえば，半々くらいの透明度にしたければ，SetLayeredWindowAttributes()でアルファ値128を指定するだけで，ウィンドウが透け透けになってしまう。

さて，問題はこのSetLayeredWindowAttributesを呼ぶ方法である。Platform SDKのライブラリをリンクすれば，なにも考えずにコールするだけなのだが，前述のようにここではそのライブラリなしでなんとかしたい。そんなことが可能だろうか？　実はライブラリはそのなかでなにか処理をしているわけではない。このSetLayeredWindowAttributes APIは，実体はシステムのUSER32.DLLのなかに含まれているのだ。ライブラリは，そのアドレスを示しているにすぎない。したがって，USER32.DLLを自前でロードし，SetLayeredWindowAttributesのエクスポートアドレスを取得すれば，ライブラリなしでもこのAPIは呼べるのだ。具体的には，

```
HISNTANCE hInst = LoadLibrary( "USER32.DLL" );
```

としてダイナミックリンクライブラリのインスタンスハンドルを取得し，

```
_SetLayeredWindowAttributes =
(BOOL(WINAPI*)(HWND,COLORREF,BYTE,DWORD))GetProcAddress( hInst, "SetLayeredWindowAttributes" );
```

SetLayeredWindowAttributes

COLUMN 2

SetLayeredWindowAttributes
レイヤードウィンドウの不透明度と透明のカラーキーを設定します。

```
BOOL SetLayeredWindowAttributes(
  HWND hwnd,          // レイヤードウィンドウのハンドル
  COLORREF crKey,     // カラーキーを指定する構造体へのポインタ
  BYTE bAlpha,        // ブレンド機能の値
  DWORD dwFlags       // アクションフラグ
);
```

パラメータ

hwnd
レイヤードウィンドウのハンドルを指定します。レイヤードウィンドウは，CreateWindowEx 関数に WS_EX_LAYERED を指定して直接作成するか，SetWindowLong 関数に WS_EX_LAYERED を指定して既存のウィンドウから作成します。

crKey
レイヤードウィンドウの作成時に使う透明のカラーキーを指定する COLORREF 構造体へのポインタを指定します。ウィンドウによって描画されるこの色のピクセルはすべて透明になります。

bAlpha
レイヤードウィンドウの不透明度を示すアルファ値を指定します。BLENDFUNCTION 構造体の SourceConstantAlpha メンバに似ています。0 を指定すると，ウィンドウは完全に透明になります。255 を指定すると，ウィンドウは不透明になります。

dwFlags
アクションフラグを指定します。次の値を組み合わせて使います。

値	意味
LWA_COLORKEY	透明色として crKey を使います。
LWA_ALPHA	bAlpha を使って，レイヤードウィンドウの不透明度を決定します。

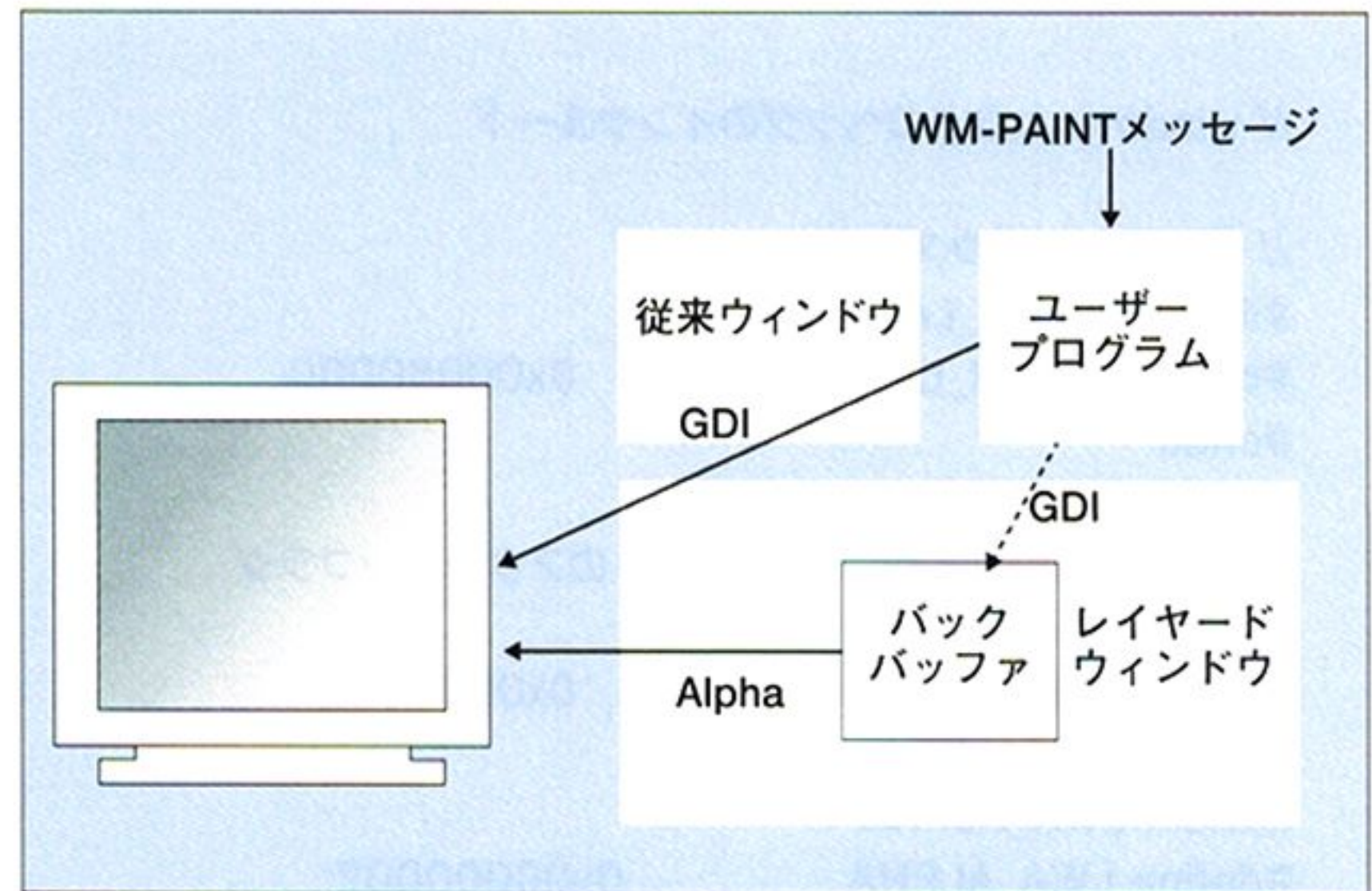



図1　高水準なレイヤードウィンドウの仕組み

としてアドレスを取得する。

この_SetLayeredWindowAttributesは，

```
BOOL (WINAPI
*_SetLayeredWindowAttributes)(HWND,COLORREF,BYTE,
DWORD);
```

としてあらかじめ宣言しておいた関数型の変数であり，このように宣言すれば，あたかも_SetLayeredWindowAttributes()という関数のようにコールすることができる。余談だがWINAPIという型はFAR PASCALのことであり，WindowsのAPIの引数はPascal渡しであることがわかる。

なお，この方法はPlatform SDKがない場合の対処法としたが，Platform SDKがある場合でもこの手法をおすすめしたい。というのは，Windows 98以前のレイヤードウィンドウ未対応OSを考えてのことだ。未対応の場合は，「このOSには対応していません」といったメッセージを出したいところだ。これにはGetVersionEx()関数でバージョンを取得し，メジャーバージョンが5.0よりも以前ならば未対応と判断すればよいだろう（Windows 2000→Windows NT 5.0)。しかし，Platform SDKのライブラリにリンクし，SetLayeredWindowAttributes()を呼んだ場合，プログラムが実行される前のメモリにロードされる際にUSER32.DLLからSetLayeredWindowAttributesを検索し，見つからない場合には「欠陥エクスポートへのリンク」エラーになってしまう。つまり，バージョンチェック部分に処理が到達することなく強制終了されてしまうのだ。

どの道，未対応の場合は動かないプログラムならばそれでも構わなくはないのだが，「対応している場合には半透明になる」アプリケーションを作りたいときは困りものだ。それに対し，自前でロードする方法ならば，まずはバージョンのチェックを行って振り分けることができるし，だいたい未対応の場合ならば_SetLayeredWindowAttributesにNULLが返るので，それで判断してもよい。

また，文字定数は以下のように設定されている。

```
WS_EX_LAYERED      0x00080000
LWA_COLORKEY       0x00000001
LWA_ALPHA          0x00000002
```

これらはPlatform SDKがあれば，そのWinUser.hの中で宣言されている。ただし，Windows 2000以降でなければ使えないように，文字定数_WIN32_WINNTおよびWINVERが0x0500以上でなければ有効にならない。そのため，以下のようにユーザーヘッダに記述しておけば，Platform SDKがある場合もない場合も面目が立つ。

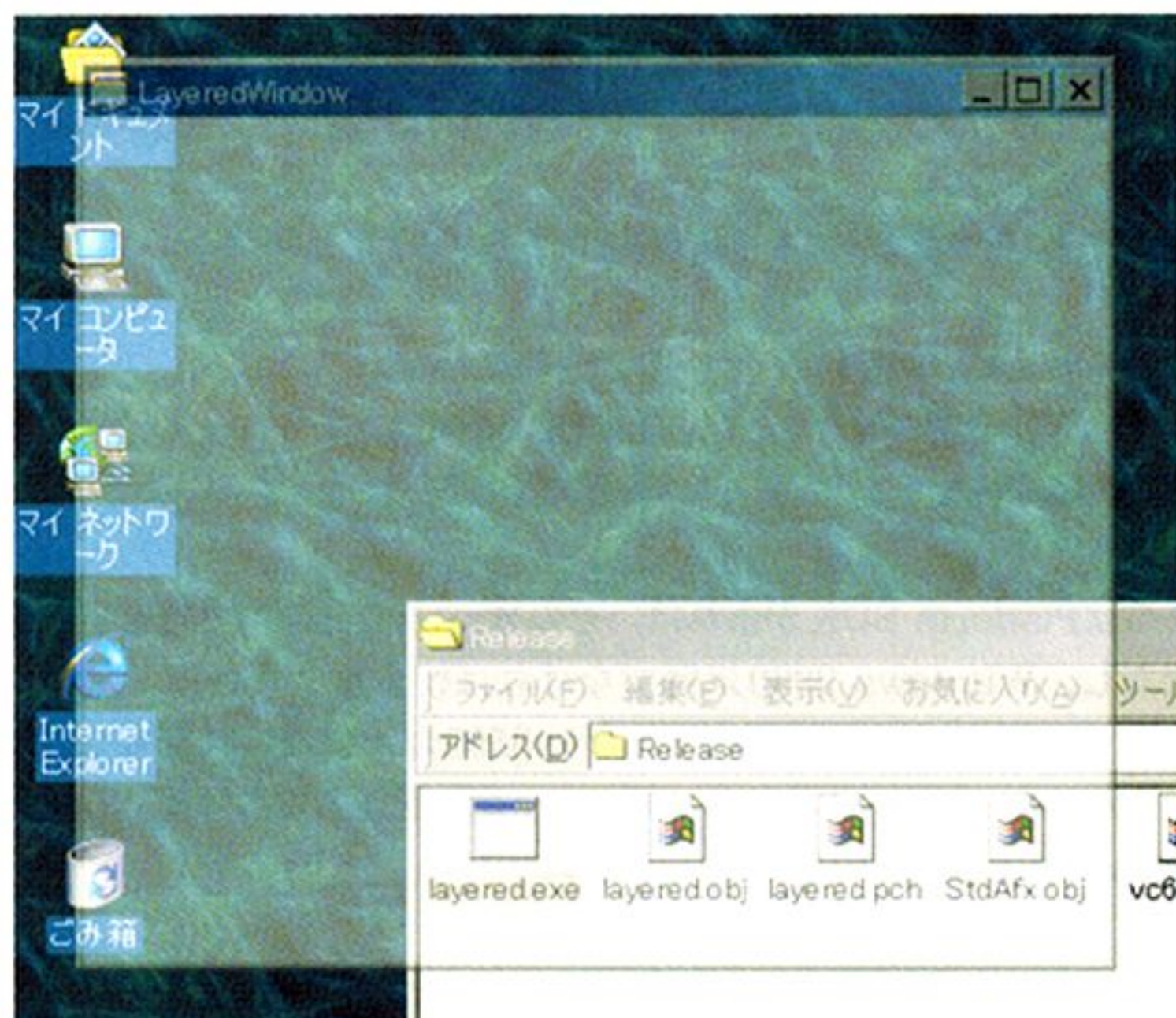
```
// Windows 2000
#define _WIN32_WINNT      0x0500
#define WINVER      0x0500
```

```
// windows.hなどのヘッダのインクルード

// 拡張ウィンドウスタイル
#ifndef WS_EX_LAYERED
#define WS_EX_LAYERED          0x00080000
#endif

// SetLayeredWindowAttributes のアクションフラグ
#ifndef LWA_COLORKEY
#define LWA_COLORKEY           0x00000001
#endif
#ifndef LWA_ALPHA
#define LWA_ALPHA              0x00000002
#endif
```

ちょっと本筋とは離れる説明が長くなってしまったが，このようにしてなんの変哲もないウィンドウを半透明にしたのが，サンプルLayered（図2）である。普通にウィンドウを開くプログラムに対して，拡張スタイルWS_EX_LAYEREDを指定し，WM_CREATEメッセージで_SetLayeredWindowAttributes()を呼んで属性にアルファ値128を設定しただけだ。このように，こちらの高水準な手法ならば，既存のアプリケーションソースにほんのちょっと手を加えるだけで半透明にできる。それどころか，拡張スタイルはウィンドウハンドルさえわかれば，あとからSetWindowLong()関数の第2引数にGWL_EXSTYLEを指定することで設定できるので，ほかのアプリケーションを外部から半透明コントロールできる。たとえば，

```
HWND hWnd = FindWindow( NULL, "電卓" );
SetWindowLong( hWnd, GWL_EXSTYLE, GetWindowLong(
hWnd, GWL_EXSTYLE )|WS_EX_LAYERED );
SetLayeredWindowAttributes( hWnd, 0, 128, LWA_ALPHA );
```

などとすれば，すでに開いている電卓を半透明にできる。なお，元に戻すときには，

```
SetWindowLong( hWnd, GWL_EXSTYLE, GetWindowLong(
hWnd, GWL_EXSTYLE )&~WS_EX_LAYERED );
RedrawWindow( hWnd, NULL, NULL,
RDW_ERASE|RDW_INVALIDATE|RDW_FRAME|RDW_ALLCH
ILDREN );
```

として拡張スタイルを外し，ウィンドウを再描画させる。

カラーキーを使った非矩形ウィンドウ

低水準な手法の前に，もうひとつ試しておこう。コラム2を見てすでに気がついているかと思うが，SetLayeredWindowAttributes()はアルファだけでなく，カラーキーも設定できる。つまり，特定の色の部分を抜くことができるのだ。抜かれた部分はヒットテストが行われず，マウスイベントもその下にあるウィンドウに透過される。これはすなわち前回やったリージョンによる非矩形ウィンドウと同じ効果となる。しかもリージョンを使った場合よりも高速で，特に外形が変化するアニメーションを行う場合などは効率的だ。カラーキーはアルファと組み合わせて使うことができるので，非矩形の半透明ウィンドウもこの手法で可能なわけだ。

ということで，前回のリージョンによる非矩形ウィンドウのサンプルを，レイヤードウィンドウで作り変えてみた。とはいっても，先ほどと基本的には変わらない。今度はウィンドウスタイルをポップアップにし（キャプションバーがない），WM_PAINTメッセージでビットマップ（図3）をデバイスコンテキストにBitBlt()する。

このデバイスコンテキストは，先ほども述べたようにバックバッファのデバイスコンテキストが渡されている。あとは（処理の順番が逆だが）WM_CREATEで，

```
_SetLayeredWindowAttributes( hWnd, RGB(0,0,0), 128,
LWA_COLORKEY|LWA_ALPHA );
```

として属性を設定する。背景部分の黒=RGB(0,0,0)をカラーキーとして抜き，さらに全体を半透明にしようというわけだ（サンプルLayered2:図4）。メニューボックスがなくなってしまったので，右クリックでコンテキストメニューが表示できるようにしてあるのは前回同様だ。このように，レイヤードウィンドウを使えば，リージョンよりも簡単に非矩形ウィンドウを作ることもできる。

UpdateLayeredWindowによる低水準な手法

高水準な手法に比べると，やや手間がかかり，既存のプログラムを対応させようと思うとそれなりの改変が必要になるが，特別難しいわけではない。拡張スタイルとしてWS_EX_LAYEREDを設定するのは同じだが，こちらはUpdateLayeredWindow()を使う（コラム3）。

この関数は，レイヤードウィンドウ（VRAMに表示されている）を直接更新するAPIだが，5番目の引数が前の説明でいうところのバックバッファの

図2　半透明ウィンドウのできあがり

図3　レイヤード用ビットマップ

図4　カラーキーを使って半透明を実現

デバイスコンテキストである。つまり，バックバッファはユーザーが自前で用意しなければならない。その代わり，バックバッファのどの領域を表示するかを指定することができる。ただし，もっとも重要なのはそれではなく，ピクセルごとにアルファを設定できることだ。第8引数で渡されるBLENDFUNCTION構造体のメンバは，次のようになっている（この構造体はVisual C++同梱のwingdi.hで宣言されている）。

```
typedef struct _BLENDFUNCTION
{
    BYTE   BlendOp;
    BYTE   BlendFlags;
    BYTE   SourceConstantAlpha;
    BYTE   AlphaFormat;
}BLENDFUNCTION,*PBLENDFUNCTION;
```

このメンバのうち，現状ではBlendOpはAC_SRC_OVER固定，BlendFlagsは0にしなければならない。SourceConstantAlphaは全体のアルファ値で0から255の範囲で設定する。SetLayeredWindowAttributesのアルファと同じことだ。最後のAlphaFormatは，現状ではAC_SRC_ALPHAフラグしかないが，このフラグこそがピクセル単位のアルファを有効にするものである。とはいえ，どこを見渡してもアルファ値を格納するバッファポインタの指定場所がない。これはどうしたらいいんだ？

BITMAP（BMP）のビットカウントは，一般的には1，4，8，24ビットである。つまり，モノクロ，16色，256色，フルカラーである。しかし，マイナーながら，16ビットや32ビットといったフォーマットも存在する。16ビットの場合はRGBそれぞれ5ビットずつの32768色，あるいはビットマスクを使って最大65536色となる。それに対し，32ビットの場合は2の32乗の色を表現できるわけではなく，実際にはRGB各8ビットの24ビットカラーとなり，残りの8ビットは遊んでいる（Windows NTでビットマスクを使えば32ビットを使い切ることは可能のようだが，あまり意味があるとは思えない）。この遊びにアルファ8ビットを入れてしまえというのは容易に想像がつくだろう。

ヘルプのBITMAPINFOHEADERの項には，「使わない」としか書かれていないが，具体的にいえば，32ビットビットマップの上位バイトは実はアルファそのものなのである。つまり，UpdateLayeredWindow()の第2引数で指定されたデバイスコンテキストに関連付けられた32ビットビットマップの最上位バイトがアルファとして認識される。

余談だが，BITMAPのヘッダはOSのバージョンとともに少しずつ拡張されてきている。一般的にはBITMAPINFOHEADERが用いられると思われるが，上位互換的にWindows 95とNT 4.0ではBITMAPV4HEADERが，Windows 98と2000ではBITMAPV5HEADERが利用できる。ヘルプのBITMAPV5HEADERの項（Platform SDKに同梱のヘルプも）にも，最上位バイトは使わないと記述されているが，そのくせbV5AlphaMaskというメンバも用意されている。アルファのビットマスクであることは明白だが，それ以上の説明はまったくなされていない。ほかの関連ヘルプを見ても，どうもピクセル単位のアルファに関してはあまり積極的に使ってほしくはないようだ。が，そんなことは知ったこっちゃない。

そんなわけで，32ビットのアルファ付きビットマップを作る必要がある。ビットマップの作成には，CreateDIBitmap()を使うのが一般的ではなかろうか。

```
HBITMAP CreateDIBitmap(
  HDC hdc,
  CONST BITMAPINFOHEADER *lpbmih,
  DWORD fdwInit,
  CONST VOID *lpbInit,
  CONST BITMAPINFO *lpbmi,
  UINT fuUsage
);
```

詳しくはヘルプに譲るが，要はlpbmihで記述されたフォーマットのビットマップを作成し，ハンドルとして返すというものだ。このとき，fdwInitにCBM_INITを設定しておけば，lpbInitで指定したデータでビットマップが初期化される。のだが，32ビットで初期データを渡しても，きっちりアルファ（最上位バイト）をゼロクリアしたビットマップが作られる。そんなに使ってほしくないかぁ！　頭にきたので，今度はCreateDIBSection()を使ってみる。

COLUMN 3　UpdateLayeredWindow

UpdateLayeredWindow
　レイヤードウィンドウの位置，サイズ，形，内容，透明度を更新します。

```
BOOL UpdateLayeredWindow(
  HWND hwnd,                // レイヤードウィンドウのハンドル
  HDC hdcDst,               // 画面のデバイスコンテキストのハンドル
  POINT *pptDst,            // 画面の新しい位置
  SIZE *psize,              // レイヤードウィンドウの新しいサイズ
  HDC hdcSrc,               // サーフェスのデバイスコンテキストのハンドル
  POINT *pptSrc,            // レイヤーの位置
  COLORREF crKey,           // カラーキー
  BLENDFUNCTION *pblend,    // ブレンド機能
  DWORD dwFlags             // フラグ
);
```

パラメータ

hwnd
レイヤードウィンドウのハンドルを指定します。CreateWindowEx 関数を使ってウィンドウを作成するとき，WS_EX_LAYERED フラグをセットすると，作成されるウィンドウはレイヤードウィンドウになります。

hdcDst
画面のデバイスコンテキスト（DC）のハンドルを指定します。このハンドルは，GetDC 関数を呼び出すときに NULL を指定することによって得られます。このハンドルは，ウィンドウの内容を更新するときパレットのカラーマッチングに使われます。このパラメータに NULL を指定すると，既定のパレットが使われます。
hdcSrc に NULL を指定するときは，このパラメータも NULL でなければなりません。

pptDst
レイヤードウィンドウの新しい画面位置が入った POINT 構造体へのポインタを指定します。現在の位置を変更しないときは，NULL を指定できます。

psize
レイヤードウィンドウの新しいサイズが入った SIZE 構造体へのポインタを指定します。ウィンドウのサイズを変更しないときは，NULL を指定します。
hdcSrc に NULL を指定するときは，このパラメータも NULL でなければなりません。

hdcSrc
レイヤードウィンドウを定義するサーフェスの DC のハンドルを指定します。このハンドルは，CreateCompatibleDC 関数または IDirectDrawSurface4::GetDC メソッドを使って取得できます。ウィンドウの形と視覚的な属性を変更しないときは，NULL を指定できます。

pptSrc
デバイスコンテキストにおけるレイヤーの位置が入った POINT 構造体へのポインタを指定します。
hdcSrc に NULL を指定するときは，このパラメータも NULL でなければなりません。

crKey
レイヤードウィンドウを構成するときに使うカラーキーが入った COLORREF 構造体へのポインタを指定します。

pblend
レイヤードウィンドウを構成するときに使う透明度の値が入った BLENDFUNCTION 構造体へのポインタを指定します。

dwFlags
次のいずれかのフラグをセットします。

フラグ	意味
ULW_ALPHA	pblend をブレンド機能として使います。ディスプレイモードが 256 色以下の場合は，ULW_OPAQUE フラグをセットした場合と同じ効果になります。
ULW_COLORKEY	crKey を透明色として使います。
ULW_OPAQUE	不透明なレイヤードウィンドウを描画します。

　hdcSrc に NULL を指定するときは，dwFlags に 0 を指定してください。

```
HBITMAP CreateDIBSection(
  HDC hdc,
  CONST BITMAPINFO *pbmi,
  UINT iUsage,
  VOID *ppvBits,
  HANDLE hSection,
  DWORD dwOffset
);
```

こちらはビットマップデータに直接アクセスできるビットマップを作る関数だ。pbmiで記述されたフォーマットのビットマップを作成し，ハンドルを返すとともに，ppvBitsにビットマップデータのアドレスを入れてくれる。このポインタを使ってデータを書き込んでやれば，ビットマップのデータを直接書き換えられるわけだ。これならば，最上位バイトをゼロクリアさせるスキを与えないはず。まったく困ったもんだ。

そんなわけで，アルファ付きの32ビットマップを作る関数を，サンプルLayered3のなかに用意した。

```
LayeredWnd::CreateAlphaContainedBitmap(
  HBITMAP hBitmap,
  HBITMAP hAlpha
);
```

hBitmapはもととなるカラービットマップ（図5）のハンドル，hAlphaはアルファを指定するビットマップ（図6）のハンドルで，そのR成分をアルファとする。グレースケールの8ビットにしておけば間違いないだろう。

実はこのビットマップを作る作業がいちばん面倒な部分だったりする。というのは，このUpdateLayeredWindow()でいったん設定してしまえば，WM_PAINTをハンドルする必要もなくなってしまうのだ。ウィンドウが移動したり，ほかのウィンドウがオーバーラップして再描画の必要が発生した場合も，システムが全自動でよきにはからってくれる。考えようによっては高水準よりも簡単だ。

ただ，これだけではナンなので，先ほどのビットマップからも想像できると思うが，アニメーションさせてみよう。これには，タイマを使う。タイマの使い方は以前やったので，ここでは詳しくは述べないが，要はSetTimer()でインターバルタイマを仕掛け，WM_TIMERメッセージをハンドルすればよい。そこでUpdateLayeredWindow()を繰り返し呼び，1コマずつ進むようにバックバッファの表示位置を更新する。いい忘れていたが，もちろんUpdateLayeredWindowもSetLayeredWindowAttributes同様，USER32.DLLから自前でロードする。なお，オプションフラグは以下のようになっている。

```
ULW_COLORKEY     0x00000001
ULW_ALPHA        0x00000002
ULW_OPAQUE       0x00000004
```

完成したのが図7（サンプルLayered3）だ。ひれの先ほど透明度を高くしてある。また，デスクトップにも影が落ちているのがわかるだろう。ただ，アルファは8ビットで指定してもあまりまじめに計算していないようで，段階的にマッハバンドが出てしまっている。また，明るい部分（背景が明るく，透明度が高い）は色が化けてしまうこともある（図8）。なんとなくオーバーフローしているような感じなのだが，どうもこの辺りがあまりピクセル単位のアルファを使わせたくない理由なのかなぁという気がする。あるいは，使える人が少ないからバグ報告もなく，フィックスされていないのかも。i810やTNT2では起こるが，GeForce2 GTSでは起こらないので，ドライバのバグという可能性が高い。

アルファに対応したGDI

そのほか，レイヤードウィンドウとは直接関係ないが，アルファ関連とし

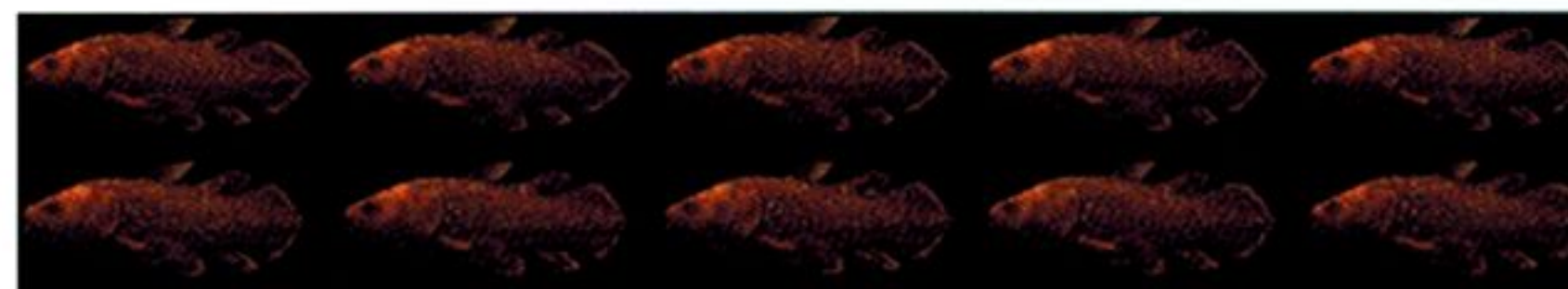



図5　カラービットマップの例

図6　アルファビットマップはこんな感じ

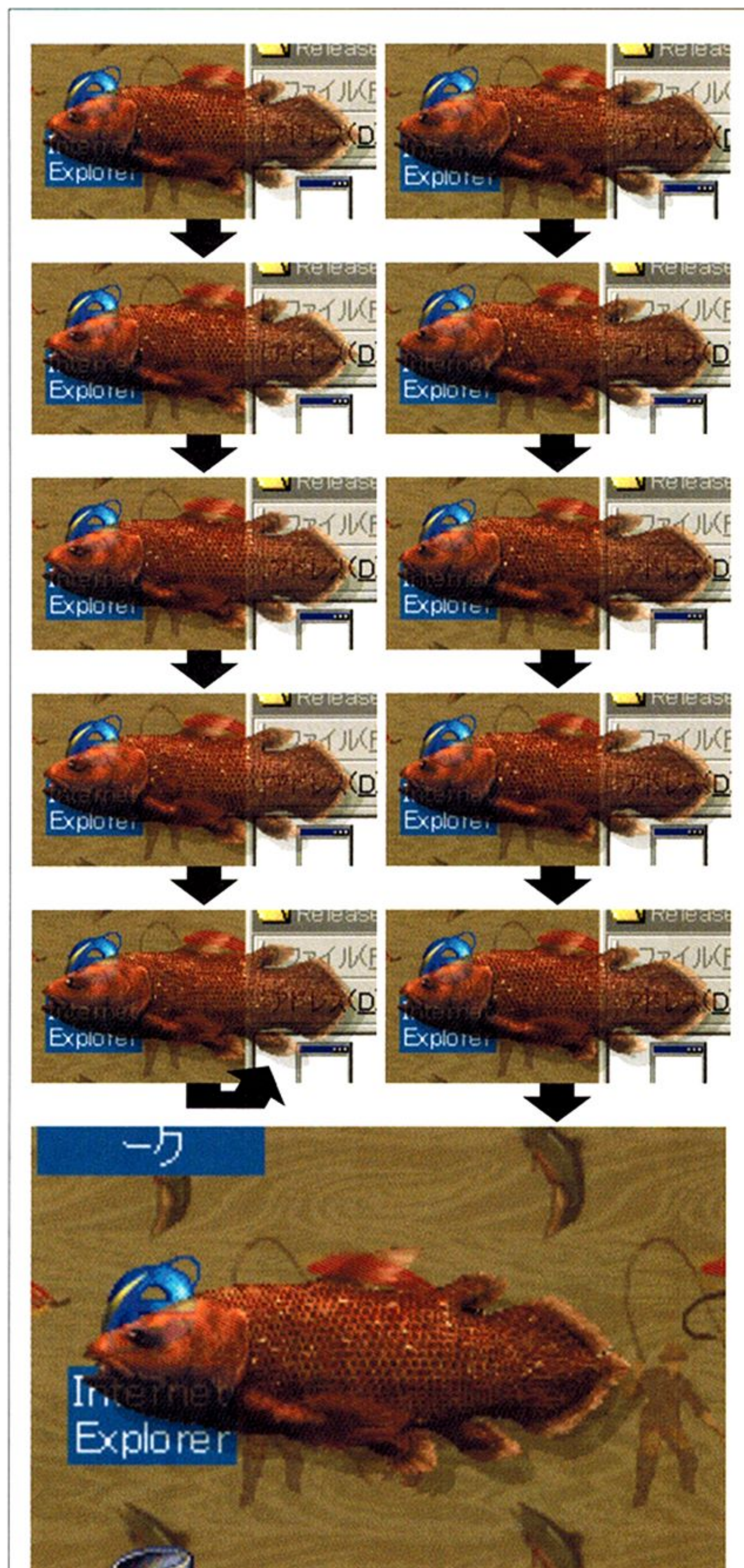



図7 シーラカンス形のウィンドウのできあがり

てAlphaBlend()やTransparentBlt()といった関数もある（隠し関数としてAlphaDIBBlend()やTransparentDIBits()といったものもあるようだ）。ピクセル単位のアルファに対応したBitBlt()（いや，StretchBlt()かな）と思えばよいのだが，こいつらはVisual C++ 6.0のヘルプにも「変更されることがある」としながら載っていて，Windows 98以上で対応と書いてある（Platform SDKのヘルプにも）。絶対そんなことはないと思いつつ，後述のオンラインドキュメントを見ると，Windows 2000以降となっている。うむうむ，そうだよなぁ。そうだと思うんだけど，試してみなきゃいかんかなぁ，ということで，AlphaBlend()をちょっと触ってみた（コラム4）。

図8　ひれの先で色化けが起きている

図9　Windows 98でアルファ付きデータをBLTしてみた

ただ，今回の本筋とは少し外れるので，とりあえずアルファが効いているかだけ試す意味で，拡張ウィンドウスタイルでWS_EX_TRANSPARENTを指定するだけの手抜きなのはご容赦願いたい。先ほどと同じ要領でアルファ付きの32ビットビットマップを作り，WM_PAINTメッセージがきたときにAlphaBlend()で貼り付けてみる。なお，AlphaBlend()を使うには，どこにも書いていないが，コンパイル時にMSIMG32.LIBをリンクする必要がある。

結果は図9（サンプルAlpha）のようになった。OSはWindows 98 SEである。激しく色化けが起こっているが，なんだ，やればできるじゃないか（ちなみにビデオはG200）。ぜんっぜん知らなかったよ。念のためにいっておくと，図を見る限りレイヤードウィンドウっぽく見えるかもしれないが，移動させると図10のように周囲を引きずってしまう，単なる初期背景が描画された矩形ウィンドウである。

そんなわけで，とりあえずPlatform SDKなしでレイヤードウィンドウを楽しめたのだが，やはり関数の詳細なヘルプがないと辛いだろう。それらはPlatform SDKにはもちろんついているのだが，ヘルプだけならMicrosoftのサイトでオンラインドキュメントを参照できる。しかもPlatform SDKのヘルプは英語だが，オンラインドキュメントは日本語化されていたりする。関連する項目のURL一覧を表1に示しておくので，参照してもらいたい。

COLUMN 4　AlphaBlend

AlphaBlend
透過ピクセルと半透過ピクセルを持つビットマップを表示します。

```
AlphaBlend(
  HDC hdcDest,
  int nXOriginDest,
  int nYOriginDest,
  int nWidthDest,
  int hHeightDest,
  HDC hdcSrc,
  int nXOriginSrc,
  int nYOriginSrc,
  int nWidthSrc,
  int nHeightSrc,
  BLENDFUNCTION blendFunction
);
```

パラメータ

hdcDest
送信先のデバイスコンテキストのハンドルを指定します。

nXOriginDest
送信先の長方形の左上角の x 座標を，論理単位で指定します。

nYOriginDest
送信先の長方形の左上角の y 座標を，論理単位で指定します。

nWidthDest
送信先の長方形の幅を，論理単位で指定します。

hHeightDest
送信先の長方形の高さのハンドルを論理単位で指定します。

hdcSrc
送信元のデバイスコンテキストのハンドルを指定します。

nXOriginSrc
送信元の長方形の左上角の x 座標を論理単位で指定します。

nYOriginSrc
送信元の長方形の左上角の y 座標を論理単位で指定します。

nWidthSrc
送信元の長方形の幅を論理単位で単位します。

nHeightSrc
送信元の長方形の高さを論理単位で指定します。

blendFunction
送信元および送信先のビットマップのアルファブレンド関数，送信元のビットマップ全体に適用されるグローバルアルファ値，送信元のビットマップのフォーマット情報を指定します。指定できる送信元および送信先のアルファブレンド関数は現在，AC_SRC_OVER に限定されています。詳しくは，BLENDFUNCTION 構造体および EMRALPHABLEND 構造体を参照してください。

図10　透明部分のアップデートに失敗している

表1

Platform SDKのダウンロード http://www.microsoft.com/msdownload/platformsdk/setuplauncher.htm
レイヤード ウィンドウ http://www.microsoft.com/JAPAN/developer/windows2000/techart/layerwin.asp
SetLayeredWindowAttributes http://www.microsoft.com/JAPAN/developer/library/jpuipf/_win32_setlayeredwindowattributes.htm
UpdateLayeredWindow http://www.microsoft.com/JAPAN/developer/library/jpuipf/_win32_updatelayeredwindow.htm
AlphaBlend http://www.microsoft.com/JAPAN/developer/library/jpgdipf/_win32_alphablend.htm
TransparentBlt http://www.microsoft.com/JAPAN/developer/library/jpgdipf/_win32_transparentblt.htm

画像可逆圧縮フォーマット「恵理ちゃん」を使う

Leshade Entis

ERI，通称「恵理ちゃん」はゲームのために作られた損失なしの画像フォーマットです。高速な展開を第一に考えられながら，写真のようなフルカラーの自然画に対してもかなりの圧縮率を誇ります。以下では恵理ちゃんの基本アルゴリズムと活用法を解説していきます。

なぜいま新フォーマットか？

問題の発端はゲームに使う3Dモデルのテクスチャ画像の形式であった。Windows Bitmap形式を利用するのはもっとも簡単な方法だが，少しサイズが大きい。JPEGのような非可逆のフォーマットは正しい選択とはいえないし……。

特にフルカラーの時代になって以来，ゲームの世界にニーズは確かにあったはずなのだが，開かれた便利な画像圧縮フォーマットは存在しなかった。PNGもそこそこいいのだが，圧縮率と展開速度が見合わないなど，ゲームに使うにはいくつかの問題もある。

なければ作る。そこで，フルカラー画像を主なターゲットとした可逆圧縮フォーマット『恵理ちゃん（ERIフォーマット）』を考案するに至ったのである。

コンセプト

新フォーマットを考案するにあたって，もっとも重点が置かれたのは，展開アルゴリズムが非常に簡単であること，そして高速に展開できるという点である。

PNGのようなネットワーク向けのフォーマットは，簡単であるということや，高速に展開できるということはあまり意味を持たない。なぜなら現時点では，ネットワークの通信速度は，ゲームで使用するCD-ROMの転送速度に比べ格段に遅く，転送から展開までに占める転送時間の割合が非常に高いからである。

しかしゲームとなると，主な情報源はCD-ROMであり，転送時間がネットワークに比べ格段に短くなるため，展開速度が重要となってくる。

もちろん，ゲームにおいても，転送から展開までの時間を小さくするためには圧縮率も関係はしてくるが，気にするのは10％の単位であり，1％単位ではない。つまり，圧縮率が40％か39％かどうかは問題ではない。むしろ問題になるのは，その1％のために増加する展開時間である。

特徴

恵理ちゃんの最大の特徴は，なんといってもその実装の簡便性であろう。展開アルゴリズムがきわめて簡単なのである。これはゲームを開発する際にはかなり重要な要素である。なぜならば，筆者からC++の展開ソースが公

表1-A　フルカラー画像圧縮比較

		ERI	ERI-A	PNG	LZH	PIC2
写真①	5,760,056 -1	2,588,908 -0.449	2,435,348 -0.423	3,498,724 -0.607	4,679,962 -0.812	2,776,161 -0.482
写真②	5,760,056 -1	2,680,854 -0.465	2,449,076 -0.425	3,889,250 -0.675	4,841,971 -0.841	3,008,035 -0.522
写真③	5,760,056 -1	2,584,236 -0.449	2,441,040 -0.424	3,442,803 -0.598	4,635,195 -0.805	2,658,892 -0.462
写真④	5,760,056 -1	2,401,437 -0.417	2,249,008 -0.39	3,595,189 -0.624	4,987,758 -0.866	2,890,352 -0.502
写真⑤	5,760,056 -1	2,191,349 -0.38	2,128,816 -0.37	3,127,045 -0.543	4,730,831 -0.821	2,534,876 -0.44
写真⑥	5,760,056 -1	2,679,589 -0.465	2,533,704 -0.44	3,656,093 -0.635	4,728,845 -0.821	2,783,847 -0.483
写真⑦	5,760,056 -1	1,936,471 -0.336	1,963,676 -0.341	2,744,770 -0.477	4,027,070 -0.699	2,306,340 -0.4
写真⑧	5,760,056 -1	1,835,874 -0.319	1,852,444 -0.322	2,247,351 -0.39	2,604,882 -0.452	1,865,850 -0.324
ＣＧ①	921,654 -1	334,321 -0.363	317,056 -0.344	373,441 -0.405	458,418 -0.497	297,456 -0.323
ＣＧ②	2,559,616 -1	1,079,803 -0.422	1,002,936 -0.392	1,196,338 -0.467	1,345,925 -0.526	971,545 -0.38
ＣＧ③	921,654 -1	232,659 -0.252	210,500 -0.228	273,852 -0.297	415,509 -0.451	264,413 -0.287
ＣＧ④	4,198,544 -1	512,076 -0.122	484,988 -0.116	532,980 -0.127	524,572 -0.125	363,585 -0.087
平均	4,556,826 -1	1,754,798 -0.385	1,672,382 -0.367	2,381,486 -0.523	3,165,078 -0.695	3,976,552 -0.873

※表中の数値はファイルのサイズ。() 括弧内の数値はBMPとの比率である。
※ERI-Aは算術符号を使ったERIファイルである。
※PNGはPhotoshopを使って圧縮した。オプションには「インターレース：なし，フィルタ：適応させる」を指定している。

表1-B　256色画像圧縮比較

	BMP	ERI	ERI-A	PNG	LZH
写真①	1,921,080 (1.00)	1,124,712 (0.585)	925,204 (0.482)	1,204,369 (0.627)	1,135,458 (0.591)
写真②	1,921,080 (1.00)	1,540,872 (0.802)	1,131,968 (0.589)	1,573,930 (0.819)	1,466,951 (0.764)
写真③	1,921,080 (1.00)	1,127,484 (0.587)	896,068 (0.466)	1,159,065 (0.603)	1,102,501 (0.574)
写真④	1,921,080 (1.00)	1,285,324 (0.669)	1,001,376 (0.521)	1,357,831 (0.707)	1,269,214 (0.661)
写真⑤	1,921,080 (1.00)	1,202,720 (0.626)	969,820 (0.505)	1,301,981 (0.678)	1,237,667 (0.644)
写真⑥	1,921,080 (1.00)	1,201,372 (0.625)	949,400 (0.494)	1,216,893 (0.633)	1,150,304 (0.599)
写真⑦	1,921,080 (1.00)	872,588 (0.454)	745,160 (0.388)	993,112 (0.517)	927,130 (0.483)
写真⑧	1,921,080 (1.00)	600,420 (0.313)	481,400 (0.251)	678,991 (0.353)	613,861 (0.320)
ＣＧ①	308,280 (1.00)	117,028 (0.380)	122,528 (0.397)	141,531 (0.459)	121,100 (0.393)
ＣＧ②	857,064 (1.00)	380,452 (0.444)	316,840 (0.370)	394,433 (0.460)	344,253 (0.402)
ＣＧ③	308,280 (1.00)	128,444 (0.417)	113,128 (0.367)	123,068 (0.399)	108,39 (0.352)
ＣＧ④	1,402,452 (1.00)	315,204 (0.225)	195,944 (0.140)	261,764 (0.187)	216,127 (0.154)
平均	1,520,393 (1.00)	824,718 (0.542)	645,586 (0.425)	867,248 (0.570)	807,747 (0.531)

※表1-Aで使用した画像をPhotoshopで誤差拡散法で256色に減色した画像を使用している。
※表中の数値はファイルのサイズ。() 括弧内の数値はBMPとの比率である。
※ERI-Aは算術符号を使ったERIファイルである。
※PNGはPhotoshopを使って圧縮した。オプションには「インターレース：なし，フィルタ：なし」を指定している。

開されているが，このソースが実用的な速度を発揮するのはPC上のみであり，現状のコンシューマ機の性能ではおそらく十分な速度を発揮できないからである。

つまり，コンシューマ機で恵理ちゃんを使用するにはチューニングが必要となるが，実装が簡単であるということはチューニングも簡単であるということである。ゲームを開発するプログラマが，ゲーム自体以外のことに割かれる時間を最小限に留める意味でもこのことは重要である。

そして，恵理ちゃんの圧縮性能も重要である。恵理ちゃんは，一般的な自然画であれば3分の1近い圧縮率を得ることができる。また，グラデーションだけでなく，アニメ絵のようなCGにも適しており，その場合，10分の1以上の圧縮率を示すこともある。

この圧縮率は，PNGと比較しても引けを取らない(多くの自然画ではPNG以上の圧縮率である)ことは，注目すべき点である(表1-A)。

編注：X68000ユーザーにはお馴染みな可逆圧縮の最高峰，柳沢PIC2と比較してもほとんど遜色はない。場合によってはERIが勝ることも多い

図1　CG①

図2　写真①

図5　写真③

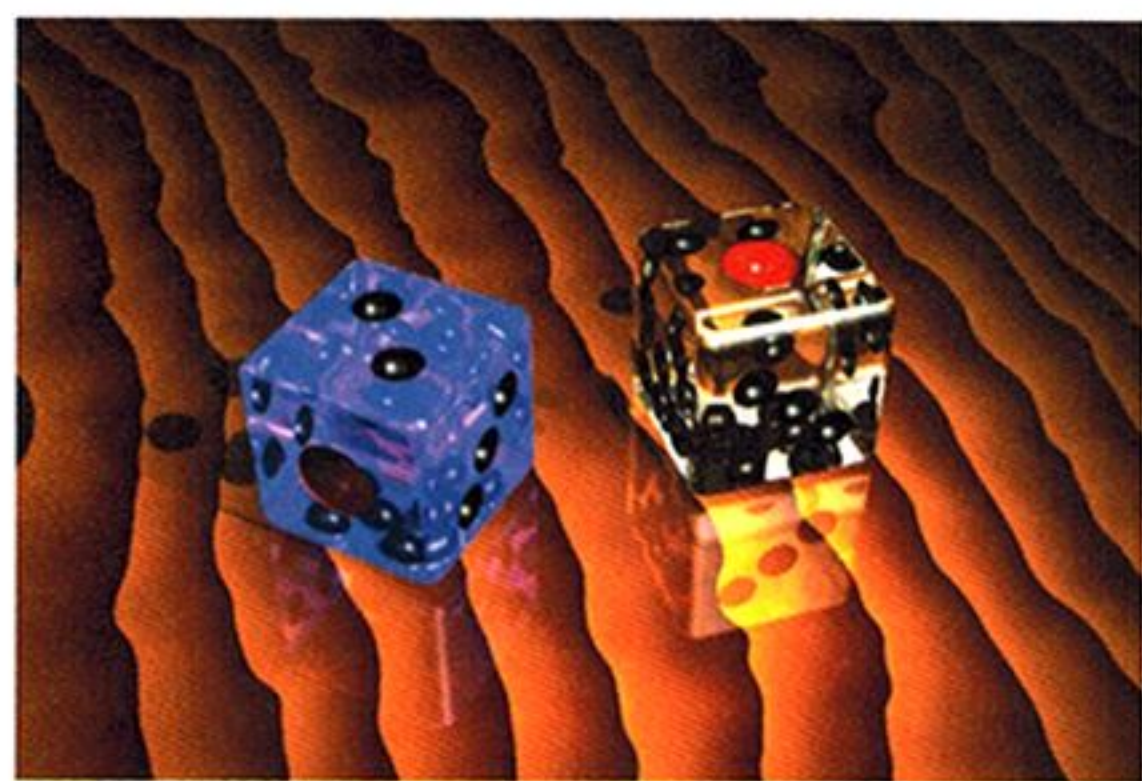



図6　CG③

図8　写真⑤

図10　写真⑥

図3　CG②

図11　写真⑦

図12　写真⑧

図9　CG④

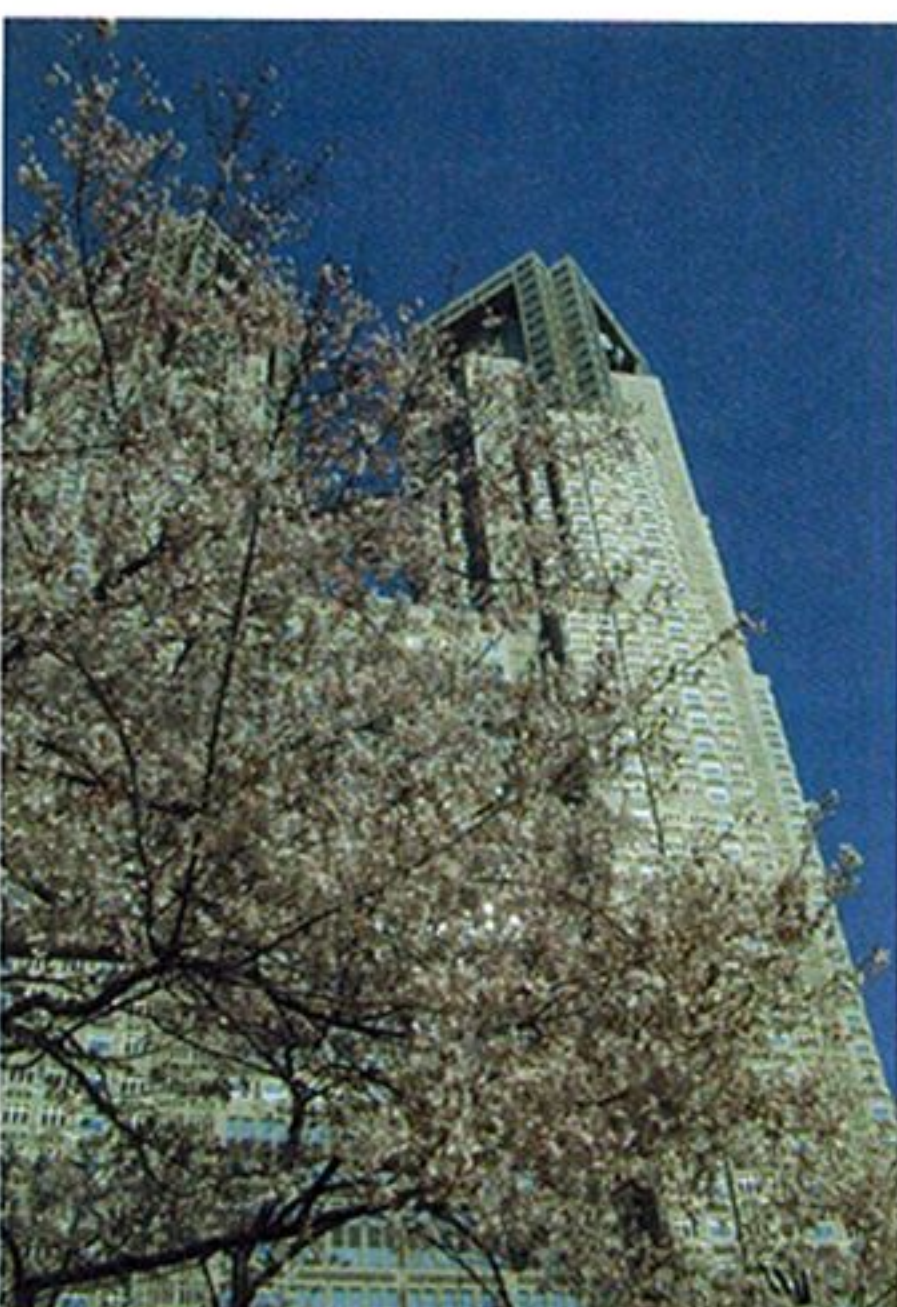



図4　写真②

図7　写真④

圧縮原理

圧縮は主に2つのフェーズから構成される。

ひとつは差分処理，もうひとつは符号化である。

一般的に自然画において，あるピクセルの近隣にはそれと近い色が存在する。差分処理のフェーズでは隣接したピクセルとの差を計算している。ただし，恵理ちゃんにおいては，フルカラーの可逆圧縮フォーマットとしては珍しく（？），チャネル間での差分処理も行っている。

もうひとつ，符号化であるが，恵理ちゃんでは独自の符号化方法を採用している。それは，ガンマ符号とランレングスを組み合わせたような符号である。

ガンマ符号とは，符号の長さを示す部分と，内容を示す部分から構成される符号で，固定確率モデルの簡易ハフマンといえるかもしれない。差分処理の結果，0近辺に値が集中しているので，値が小さいほど少ないビットを割り当てる構成になっている。また，0が連続する場合には，その連続する数を指定する方法を取っている。

差分処理

それでは，詳しく圧縮方法を見ていこう。

圧縮では初めに，空間2次元差分処理を施す。これは画像のサイズをw×hピクセルとしたとき，2次元の情報源s(i,j)に対して次の処理を施す。

(1) $s'(i,j) = s(i,j) - s(i-1,j)\ (i=1,\cdots(h-1))$
(2) $s''(i,j) = s'(i,j) - s(i,j-1)\ (j=1,\cdots(w-1))$

この，(1)，(2)の処理は，順序が逆でも得られる結果は同じである。

この2次元差分処理は，空間の2次元に，各ピクセルの輝度をあわせた3次元空間上で，特定のピクセルについて，そのピクセルの左上，上，左隣のピクセルを含む平面上の点を予測値とした場合の差を算出する計算にほかならない。

次に，n×nピクセルのブロックに分解する。そして，各ブロック内で，カラーオペレーションというチャネル間での差分処理を施す。

カラーオペレーションにはいくつかの種類があり，もっとも有効なチャネル間での差分処理を選択して実行する（表2）。

RGB-YUV変換のような色空間変換では可逆性が失われるので，カラーオペレーションは，単純なチャネル間の減算処理として定義されており，ブロック内部の色の偏りに対応できるようになっている。ブロックにあらかじめ分解するのは，空間的な色の偏りを圧縮により効率的に利用するためである。

ランレングスーガンマ符号

この符号は，0の塊と，0でない信号の塊に分け，それを交互に配列した形状を取る。ひとつの信号の塊はその長さと，0でない信号の塊の場合には，いくつかの信号の配列から構成される。

たとえば，次のような情報があったとしよう。

－1, 0, 0, 0, 3, 6, －1, －4, 0, 0

これを，0と0でない信号の塊に分けると次のようになる。

{－1}, {0, 0, 0}, {3, 6, －1, －4}, {0, 0}

そして，それぞれの塊の長さを付け加えて，0を除くと次のようになる。

1{－1}, 3, 4{3, 6, －1, －4}, 2

この数値の配列をガンマ符号化する。

このガンマ符号は0を表現する必要がないので，1を1ビットで，2～3を3ビットで，4～7を6ビットでというふうに表現できる（表3）。

アルファチャネルについて

恵理ちゃんでは，アルファチャネルつきの画像を保存することができる。

アルファチャネルとは，各ピクセルごとの不透明度を指定するチャネルのことであり，フルカラーゲームには欠かせない存在である。

ところで，アルファチャネルつきの画像データの格納方法には2種類ある。結合型と非結合型である。

結合型とは，RGBチャネルが，アルファチャネルで乗算された値で格納されている方法である。恵理ちゃんでは結合型で画像を保存することが推奨されている（ちなみにPNGでは非結合型と規定されている）。

これには次のような理由がある。

まず第一に，あらかじめRGBチャネルがアルファチャネルによって乗算されているので，実際にアルファチャネルを使って描画する際，演算量が減少するという利点がある。

これは，アルファブレンディング（半透明描画）に必要となる式を見ればわかるだろう。

$$d' = d * (1 - a) + s * a$$

aは不透明度，sは描画する画像の輝度，dは描画される画像の輝度であり，すべて0から1の範囲の値を取るものとする。

右辺の第2項に注目していただきたい。仮に，

$$s' = s * a$$

表2　カラーオペレーション

コード	格納データフォーマット
0000	（B，G，R，A）
0001	未定義
0010	未定義
0011	未定義
0100	未定義
0101	（B，G－B，R，A）
0110	（B，G，R－B，A）
0111	（B，G－B，R－B，A）
1000	未定義
1001	（B－G，G，R，A）
1010	（B，G，R－G，A）
1011	（B－G，G，R－G，A）
1100	未定義
1101	（B－R，G，R，A）
1110	（B，G－R，R，A）
1111	（B-R，G－R，R，A）

※B, G, R, Aは青、緑、赤、アルファの各チャネルの輝度を表しており、対応コードの減算の組み合わせを示している。

表3　ガンマ符号

ガンマ符号	値の範囲
0	1
1X0	2～3
1X1X0	4～7
1X1X1X0	8～15
1X1X1X1X0	16～31
1X1X1X…0	2^n～$2^{n+1}－1$

のように定義すると,

$$d' = d * (1 - a) + s' \quad \cdots\cdots ①$$

となることがわかるだろう。s'とは，結合型で保存されるRGBチャネルの値そのもののことである。

つまり，必要となる乗算をあらかじめ行っておくことにより，実行時間に乗算する回数を減らしていることになる。

ほかにも結合型の利点として，結合された画像の描画関数を加算描画関数としても使えるという点もある。

式①を見れば，描画する画像データを単に加算しているということに気づいていただけるだろう。aが0のときは完全に単純な加算処理となる。

また，利点というほどのものではないが，結合型であるということは，画像データを作る際の問題もある程度解決している。

というのも，Photoshopなどで透明度も含む画像を恵理ちゃんで保存するとしよう。結合型であれば，単純に黒い背景と合成してRGBチャネルとして出力すればそれでよい。しかし，非結合型ではそうはいかない。

こういった実用面からゲームとしては結合型が向いており，恵理ちゃんも結合型データであることが推奨されている。

256色フォーマット

恵理ちゃんはフルカラー画像だけではなく，のちに正式に256色画像にも対応した。フォーマットは両者で異なるが，フルカラー画像で使用しているガンマ符号をベースにしたフォーマットなので，ほんのわずかなコードの追加で256色にも対応できるのが特徴である。圧縮率はPNGには多少劣るものの，十分に実用レベルである。

ファイルフォーマット

ERIファイルは，64バイトのファイルヘッダと，それに続く構造化されたチャンク（ERIファイルではレコードと呼んでいる）から構成されている。

レコードの構成は下図のようになっている。

ストリームレコードには，複数の画像データを格納できるので，アニメーションデータとして作成することも可能である。

恵理ちゃんを使おう☆

では，恵理ちゃんをゲームで使うにはどうしたらよいだろうか？

恵理ちゃんの公式サイトである恵理ちゃんclub（http://www.entis.gr.jp/eri/）から恵理ちゃんの展開プログラムが配布されているので，それを利用するのがもっとも簡単な方法である。

プラットホームがWin32であれば，Win32専用の高速化されたライブラリを使うことができる。汎用のC++ソースもあるので，基本的にはプラットホームは問わない。

まず，ライブラリをダウンロードしよう。ライブラリには，ドキュメントとヘッダ，スタティックライブラリだけでなく，改造ができるようにソース，またサンプルとしてSusieプラグインのソースが添付されている。

プログラムの手順

恵理ちゃんの展開の手順は次のとおりである。

(1) ERIDecoderオブジェクトを構築する
(2) ERIDecoder::Initialize関数でデコーダを初期化する
(3) ERIDecoder::DecodeImage関数で画像を展開する

ERIDecoder::DecodeImage関数には，RASTER_IMAGE_INFO構造体で出力先の画像バッファとRLHDecodeContextオブジェクト，そして展開をトップダウン形式で行うか否かを指定する。RLHDecodeContextクラスは，入力する画像データへのインタフェイスを供給する。

アプリケーション作成者はRLHDecodeContextクラスを派生させ，ReadNextData関数をオーバーライドし，圧縮されたデータへのインタフェイスを供給しなければならない。

ReadNextData関数のプロトタイプは次のようになっている。

```
virtual ULONG ReadNextData
( PBYTE ptrBuffer, ULONG nBytes ) ;
```

ひとつ目の引数はデコーダが必要としている次のデータを格納すべきバッファへのポインタを示している。2つ目の引数はバッファのバイト数である。

アプリケーション作成者はこの関数をオーバーライドし，ファイルなどからデータを読み込み，実際に読み込まれたバイト数を返せばよい。また，ライブラリには特別にファイルを読み込むためのクラスも用意されている。

これらは，EFileObject抽象クラスからの派生クラスで，ファイルへのインタフェイスを供給するERawFileクラス，メモリ上に展開されたデータに対するインタフェイスを供給するEMemFileクラス，ERIファイルに対するインタフェイスを供給するERIFileクラスである。

たとえば，ディスク上に保存されている"sample.eri"というファイルにアクセスするには，まずERawFileクラスでそのファイルを開く（ERawFile::Open関数）。次に，ERIFileクラスでERIファイルに対するインタフェイスを開く（ERIFile::Open関数）。

これは次のようなプログラムになる。

```
ERawFile    rf ;
ERIFile     erif ;
if ( !rf.Open( "sample.eri" ) )
{
        // エラー処理
}
if ( !erif.Open( &rf ) )
{
        // エラー処理
}
```

ERIFile::Open関数はヘッダを読み込み解析し，正しいERIファイルであった場合，画像データレコードを開いてtrueを返す。

ERIFile::Open関数を呼び出したあと，ERIFileクラスを通じたファイルへのアクセスは，画像データレコード内に限定される。たとえば，ERIFile::GetLength関数はファイルの全長ではなく，画像データのバイト数を返す。

また，ERIFile::Open関数ではファイルのヘッダ情報が次のメンバ変数に格納される。

m_FileHeader:ERI_FILE_HEADER
ファイルヘッダレコードの内容が読み込まれている。

m_InfoHeader:ERI_INFO_HEADER
画像情報ヘッダレコードの内容が読み込まれている。
ERIDecoder::Initialize関数には，この構造体を渡せばよい。

m_PaletteTable[0x100]:ENTIS_PALETTE
パレットテーブルが読み込まれている。

m_strCopyright:EString obj
著作権情報が読み込まれている。

m_strDescription:EString obj
コメントが読み込まれている。

より高度な利用方法

ERIDecoderクラスには，仮想関数がいくつか定義されており，それら

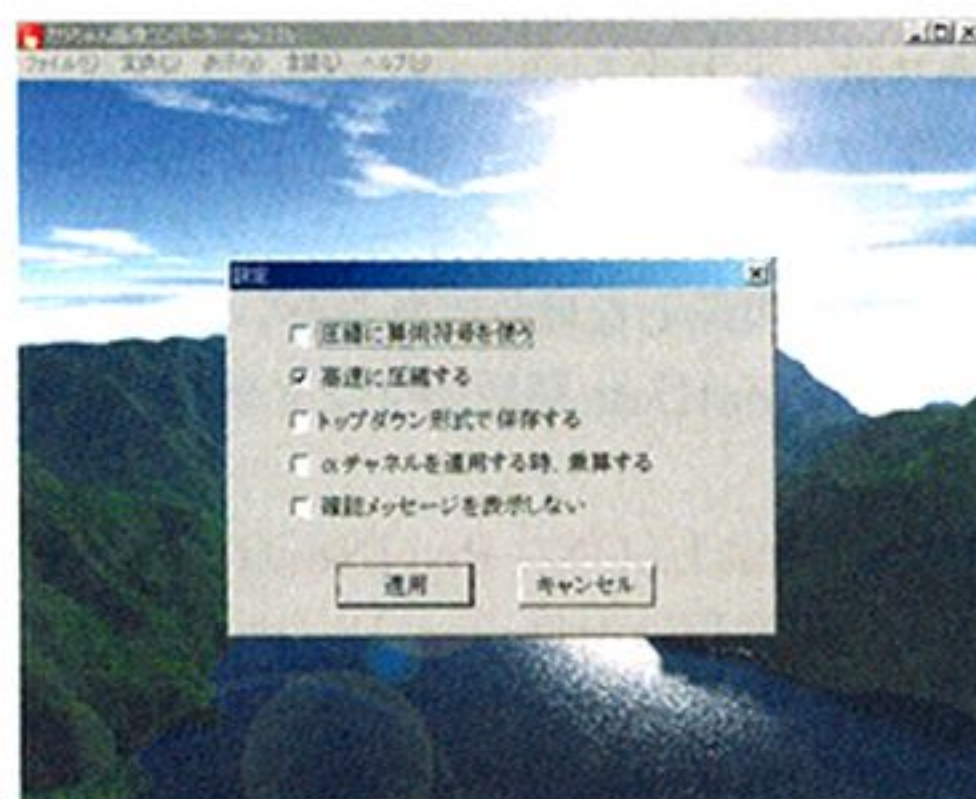



図13 理恵ちゃん画像コンバータ

の関数をオーバーライドすることによってカスタマイズすることができる。

ビュアなどを作る際には画像の展開の進行状況を表示する必要が出てくるが，そのためにはOnDecodedBlock関数とOnDecodedLine関数をオーバーライドすればよい。これらの関数は1ブロック，または1ライン展開されるごとに呼び出される。

また，環境によっては，またはなんらかの必要に応じて特殊なピクセルのフォーマットで展開したいかもしれない。そのためにはGetRestoreFunc関数をオーバーライドする。この関数は圧縮されている画像のフォーマットを受け取り，展開されたデータを出力バッファにストアする関数へのポインタを返す。派生したクラスで画像のストア関数を書き，GetRestoreFunc関数で，そのストア関数へのポインタを返せばブロックごとにそのストア関数が呼び出されるようになる。

これらの詳細な情報は恵理ちゃんclubやライブラリに同梱されているドキュメントを参照していただきたい。

恵理ちゃんの将来

恵理ちゃんは先に述べたように，別にPNGに対抗して考案されたフォーマットというわけではないのだが，そのように見られることが多い。

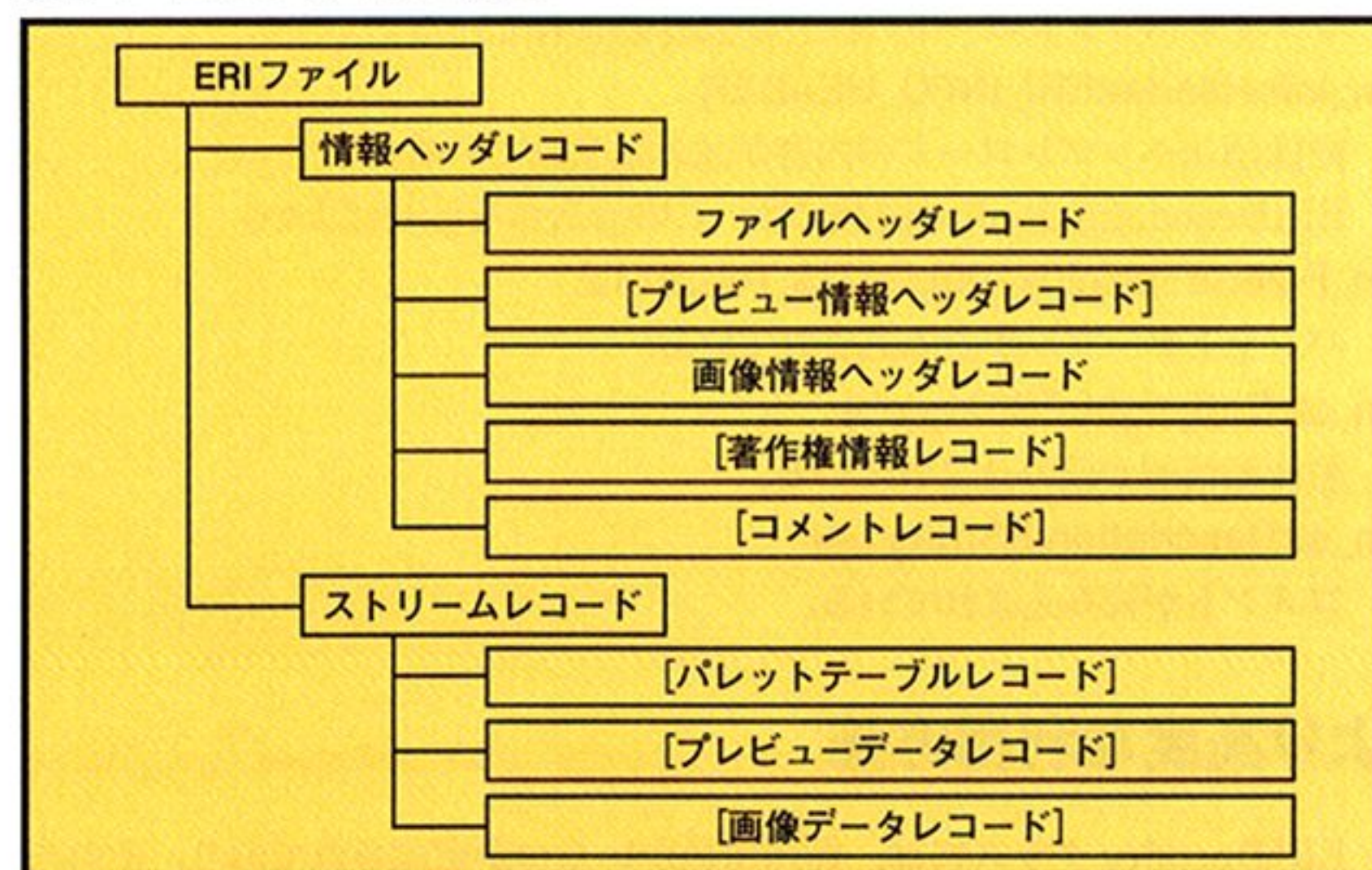
だからというわけではないが，ゲームではあまり使われないであろう算術符号を使った圧縮オプションが用意されており，これで圧縮すると展開速度は遅くなるものの，さらに高い圧縮率を得ることができる。このオプションは明らかにネットワークに使用されることを視野に入れたオプションである。ネットスケープ用のプラグインをリリースしていることからもそれは明らかであろう。

しかし，やはりメインターゲットはゲームである。

最近はインターネットを使って自分の作ったゲームを公開するというアマチュアゲームクリエータも多いと思うが，その場合にも恵理ちゃんは強力な味方となるだろう。恵理ちゃんは普通の画像であればLZHよりも圧縮率が高い。当然，LZHに圧縮された恵理ちゃんは，LZHに圧縮されたビットマップファイルよりも小さい。

この先，ゲーム開発用のツールなどに恵理ちゃんが採用されていけば，ゲーム開発の際の標準的な画像形式になりうる可能性を秘めている。

図14 ERIファイル構造

- ERIファイル
  - 情報ヘッダレコード
    - ファイルヘッダレコード
    - [プレビュー情報ヘッダレコード]
    - 画像情報ヘッダレコード
    - [著作権情報レコード]
    - [コメントレコード]
  - ストリームレコード
    - [パレットテーブルレコード]
    - [プレビューデータレコード]
    - [画像データレコード]

リスト

```
---------- eri2bmp.cpp

/*****************************************************************************
            恵理ちゃん → ビットマップ 変換プログラム
                        2000/11/27
------------------------------------------------------------------------------
            Copyright (C) 2000 Leshade Entis. All
                    rights reserved.
 *****************************************************************************/

//
// ヘッダの読み込み
//////////////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <memory.h>

typedef   signed short int    SWORD ;
typedef   signed long int     SWORD ;
typedef   unsigned __int64  UINT64 ;

#include "experi.h"


//
// 入力コンテキストの宣言と定義
//////////////////////////////////////////////////////////////////////////////

class MyDecodeContext     : public          RLHDecodeContext
{
ERIFile * m_pfile ;

public:
// 構築関数
MyDecodeContext( ERIFile * pfile )
   : RLHDecodeContext( 0x1000 ), m_pfile( pfile ) { }
   // データ入力関数
virtual ULONG ReadNextData( PBYTE ptrBuffer, ULONG nBytes ) ;

;
}
ULONG MyDecodeContext::ReadNextData( PBYTE ptrBuffer, ULONG
nBytes )
{
   return m_pfile->Read( ptrBuffer, nBytes ) ;

}

//
// メモリアロケーション
//////////////////////////////////////////////////////////////////////////////

PVOID eriAllocateMemory( DWORD dwBytes )
{
   return malloc( dwBytes ) ;
}

void eriFreeMemory( PVOID pMemory )
{
   free( pMemory ) ;
}


//
// エントリポイント
//////////////////////////////////////////////////////////////////////////////

int main( int argc, char * argv[] )
{
   //
   // プログラム見出し表示
   //
   printf( "恵理ちゃん → ビットマップ 変換プログラム¥n" ) ;
   printf( "Copyright (C) 2000 Leshade Entis. All rights reserved.¥n" ) ;
printf( "¥n" ) ;

if ( argc < 3 )
{
   //
   // 書式表示
   //
   printf( "書式 : eri2bmp <ERIファイル> <BMPファイル>¥n" ) ;
   printf( "¥n" ) ;
   printf( "   ERIファイル : 変換するERIファイル名を指定します。¥n" ) ;
```

```
        printf( "    BMPファイル：変換後のBMPファイル名を指定します。¥n" ) ;
        printf( "¥n" ) ;
        return  0 ;
    }

    //
    // ERIファイルを開く
    //
    EReadFile     rf ;
    if ( !rf.Open( argv[1] ) )
    {
        printf( "¥"%s¥" を開けませんでした。¥n¥n", argv[1] ) ;
        return  1 ;
    }
    ERIFile    erif ;
    if ( !erif.Open( &rf ) )
    {
        printf( "¥"%s¥" は恵理ちゃんではありません。¥n¥n", argv[1] ) ;
        return  2 ;
    }

    //
    // ビットマップを展開するための準備
    //
    RASTER_IMAGE_INFO  rii ;
    rii.fdwFormatType=erif.m_InfoHeader.fdwFormatType ;
    rii.nImageWidth=erif.m_InfoHeader.nImageWidth ;
    rii.nImageHeight=abs(erif.m_InfoHeader.nImageHeight ) ;
    rii.dwBitsPerPixel=erif.m_InfoHeader.dwBitsPerPixel ;
    rii.BytesPerLine=
        ((rii.nImageWidth rii.dwBitsPerPixel + 0x1f)
                  & ~0x1f) >> 3 ;
    rii.ptrImageArray =
        (PBYTE) malloc( rii.BytesPerLine * rii.nImageHeight ) ;
    //
    printf( "現在展開中です・・・" ) ;

    //
    // 展開開始
    //
    ERIDecoder  decoder ;
    MyDecodeContext   context( &erif ) ;
    if ( decoder.Initialize( erif.m_InfoHeader ) )
        {
        printf( "初期化に失敗しました。¥n" ) ;
        free( rii.ptrImageArray ) ;
        return  3 ;
    }
    if ( decoder.DecodeImage( rii, context, false ) )
    {
        printf( "失敗しました。¥n" ) ;
        free( rii.ptrImageArray ) ;
        return  4 ;
        }
    //
    printf( "終了しました。¥n" ) ;

    //
    // BMPファイルを開く
    //
    FILE * bmpf = fopen( argv[2], "wb" ) ;
    if ( bmpf == NULL )
    {
        printf( "¥"%s¥" を開けませんでした。¥n¥n", argv[2] ) ;
        free( rii.ptrImageArray ) ;
        return  5 ;
    }

    bool   fError = false ;
    printf( "現在書き出し中です・・・" ) ;
    do
    {
        //
        // ビットマップファイルヘッダを書き出す
        //
        int  pltlen = 0 ;     // パレットの長さ
        if ( rii.dwBitsPerPixel <= 8 )
        {
        pltlen = (1 << rii.dwBitsPerPixel) ;
        }
        BITMAPFILEHEADERbmfh ;
        bmfh.bfType = *((WORD *)"BM") ;
        bmfh.bfOffBits = sizeof(BITMAPFILEHEADER)
                    +  sizeof(BITMAPINFOHEADER)  +  sizeof(RGBQUAD) *
pltlen ;
        bmfh.bfReserved1 = 0 ;
        bmfh.bfReserved2 = 0 ;
        bmfh.bfSize = bmfh.bfOffBits + rii.BytesPerLine * rii.nImageHeight ;
        if ( fwrite( &bmfh, sizeof(bmfh), 1, bmpf ) < 1 )
        {
            fError = true ;
            break ;
        }
        //
        // ビットマップ情報ヘッダを書き出す
        //
        BITMAPINFOHEADER    bmih ;
        memset( &bmih, 0, sizeof(bmih) ) ;
        bmih.biSize = sizeof(bmih) ;
        bmih.biWidth = rii.nImageWidth ;
        bmih.biHeight = rii.nImageHeight ;
        bmih.biPlanes = 1 ;
        bmih.biBitCount = (WORD) rii.dwBitsPerPixel ;
        bmih.biCompression = BI_RGB ;
        bmih.biSizeImage = rii.BytesPerLine * rii.nImageHeight ;
        if ( fwrite( &bmih, sizeof(bmih), 1, bmpf ) < 1 )
        {
        fError = true ;
        break ;
        }
        //
        // パレットテーブルを書き出す
        //
        if ( pltlen > 0 )
        {
            RGBQUAD     rgbTable[0x100] ;
            if ( rii.fdwFormatType & ERI_WITH_PALETTE )
            {
                memcpy( rgbTable,
                erif.m_PaletteTable, sizeof(RGBQUAD) * pltlen ) ;
            }
            else
            {
                BYTE  step = (BYTE)(1 << (8 - rii.dwBitsPerPixel)) ;
                BYTE  value = 0 ;
            for ( int i = 0; i < pltlen; i ++ )
                {
                rgbTable[i].rgbBlue=value ;
                rgbTable[i].rgbGreen=value ;
                rgbTable[i].rgbRed = value ;
                rgbTable[i].rgbReserved = 0 ;
                value += step ;
            }
            if ( fwrite( rgbTable,
                sizeof(RGBQUAD), pltlen, bmpf ) < (size_t) pltlen )
            {
            fError = true ;
                break ;
            }
        }
            //
            // ビットマップ配列を書き出す
        //
        if ( fwrite( rii.ptrImageArray, bmih.biSizeImage, 1, bmpf ) < 1 )
        {
            fError = true ;
            break ;
        }
    }
    while ( false ) ;
    //
    fclose( bmpf ) ;
    //
    if ( fError )
    {
        printf( "失敗しました。¥n" ) ;
        free( rii.ptrImageArray ) ;
        return  6 ;
    }
    printf( "終了¥n¥n" ) ;

    //
    // 後始末
    //
    free( rii.ptrImageArray ) ;

    return 0 ;
}
```

FutureBASIC³での文字コード変換の実装
～Text Encoding Converterを使ってみる

水野貴明　Mizuno Takaaki

MacintoshのFutureBASICを使ったプログラム入門です。ここではC言語用に用意されたAPIをFutureBASICから呼び出してみます。これによって，OSやほかのシステムによって用意されたさまざまな機能が使えるようになるのです。今回は文字コード変換を行ってみます。

バベルの塔の崩壊以後，世界中にはものすごくたくさんの言語と文字が生まれてしまいました。しかたがないので，世界中のコンピュータはそれぞれそのユーザーの使用している言語にあわせて，さまざまな文字が使えるように工夫されています。また，同じ言語を操る人同士でも，コンピュータ環境はそれぞれに異なっています。ネットワーク技術の進歩とともに，アプリケーションにおいてそのような環境の違いによる「文字化け」を防ぐための文字コード変換の必要性はますます高まってきました。そこで今回はFutureBASIC³（以下FB³）を用いて，文字コード変換機能を簡単に実装する方法について考えてみたいと思います。

文字コードとは？

コンピュータはあまり賢くないので基本的には数字しかわかりません。というより1と0しかわかりません。それは，電気信号のオンとオフで処理が行われているからです。しかし，人間様は文字という便利なものを持っていて，それをコンピュータにもわかってほしいと思ったのです。そこで，使いたい文字や記号にそれぞれ固有の数字を割り当て，その数字を並べることで文字を表すことにしました。これが「文字コード」と呼ばれるものです。欧米のような使っている文字数が少ない諸言語では，8ビット（1バイト），すなわち256文字でほとんどの文字を表すことができるのですが，日本語や韓国語，中国語のように漢字を使う民族は，そうはいきません。そこで16ビット（2バイト）を使って漢字を表したりしています。これが日本語の文字が「2バイトコード」などと呼ばれる所以です。

文字コードと呼ばれるものは，実際には「文字集合（Character Set）」と「符号化方式（Character Encoding）」という2つの概念でできあがっています。文字集合とは，世界中に無数にある文字のうち，どの文字とどの文字を使うかを決めた「使う文字一覧」のことです。それを実際にどのような数値に当てはめて使うか，ということを決めることを「符号化」といい，通常文字コードといわれるものはすべてこの符号化の方式を決めた「符号化方式」というものに属しています。

主な文字コードの種類と仕組み

日本で生活し，日本語を話し，日本語のOSの搭載されたコンピュータを使っていると，よく耳にする文字コードは「SHIFT-JIS」「JIS」「EUC-JP（以下，EUC）」の3つだと思います。これらの3つはすべて符号化方式を指す言葉なのです。そしてこの3つの符号化方式は同じ文字集合を用いています。つまりこれらのコードで表すことができる文字の種類は同じということです。その文字集合は「JIS X 0201」と「JIS X 0208」です。「JIS X 0201」は1バイトで表現する文字を決定した文字集合（いわゆるJISラテン文字と半角カナ）で，「JIS X 0208」は2バイトの文字を定義した文字集合（いわゆる区点コード）です。

EUCとJISの場合はさらに，「JIS X 0212」といういわゆる「補助漢字」と呼ばれる文字集合も含んでいます。3つのコードがすべて同じ文字集合を使用しているので，実際のコードが異なっていても，簡単な変換式で相互に変換が可能になっているわけです。これらの文字集合は「JIS」という言葉からもわかるように，すべて日本工業規格で策定されています。

なお，JIS規格に関しては，2000年2月に「JIS X 0213」という新たな文字集合が定義されました。このなかには現在日本で名前などに使われている異体字などがほぼ網羅されています。しかし，牛丼の吉野屋などで使われている上が「土」になっている「吉」の字や「高」の異体字で「梯子高」などが入っていないなど，まだ問題を含んでいるようです。

JISラテン文字はASCIIというアメリカで決められた文字集合をもとに作られたものです。ASCIIは，7ビットですべての文字を表すことができるのですが，なにしろアメリカで決められたものだけに，アメリカ人以外の人が使うと，不便な点がいくつもありました。たとえば貨幣を表す記号が「$」しかなかったり，ヨーロッパの言語によくある，アルファベットに発音記号をつけたようなものも使えません。そこで各国でさまざまな変更・拡張が行われました。JISラテン文字もそのひとつです。

その変更の方法はISO 646という国際規格に則っています。これはASCII文字集合のうちどの部分は変えてもよくて，どの部分は変えてはいけないかを規定したものです。JISラテン文字では「＼（バックスラッシュ）」を「￥」，「~」を「￣」という文字で置き換えています。

さて「符号化方式」のほうのJISコードは，junetコードとも呼ばれ，正確には「ISO-2022-JP」と呼ばれる符号化方法です。これはISO 2022という国際規格のサブセットで，「モード切り替え方式」と呼ばれる方式を採用しています。モード切替方式というのは，複数の重複したコード領域を持つ文字集合を，制御コードを使って切り替えながら使う方法のことです。この方式は制御コードがたくさん埋め込まれるためにデータサイズも大きくなり，処理も面倒くさくなるので，特に昔のような処理速度が遅くてメモリも少なかったコンピュータでは，あまり具合がよくありませんでした。

そんなおり，1983年にマイクロソフトがJIS X 0201で使われていない領域にJIS X 0208のデータをずらして（＝シフト）詰め込んでしまえばいいのでは，と考えついたのです。そして生まれたのがSHIFT-JISという符号化方式でした。SHIFT-JISはWindowsやMacintosh，HP-UXなどの一部のUNIX系OSでも採用されており，1997年にはJIS X 0208の定義にも追加され，現在でももっとも一般的な文字コードとになりました。

EUCは正確には日本語EUCという，AT&Tによって開発されたUNIX拡張コード（Extended Unix Code）の日本語版です。UNIX系OSでよく使われているコードで，日本語のほかにも韓国語版や中国語版が存在します。

日本でメジャーな文字コードは以上の3つですが，海外では，それぞれの国で，それぞれの文字のために考え出された文字コードがたくさん存在し，日々プログラマたちを悩ませています。たとえばヨーロッパではISO-8859という文字コードが一般的です。これはASCIIを8ビットに拡張し，ヨーロッパ言語で用いられる発音記号や特殊記号を割り当てたもので，「ISO-8859-1」から「ISO-8859-15」までの15種類あります。ISO-8859-1は，別名Latin-1と呼ばれるもので，西欧諸国の特殊文字が定義されています。ISO-8859-2は東欧諸国の特殊文字を集めたものです。

Text Encoding Converterとは

Text Encoding Converter（以後TEC）は，MacOS 8から標準でインストールされるようになった共有ライブラリで，さまざまな言語で用いられている文字コードのチェックや変換などを行うためのものです。System7で

は標準ではついてきませんが，SDKについてくるドキュメントによると，7.1以上であればきちんと動くようです。このライブラリを使えば，自分で文字コード変換ルーチンを実装しなくても，文字列変換をプログラム中で行うことができます。

システムフォルダを開くと，「機能拡張」フォルダに「Text Encoding Converter」というライブラリファイル(図1)が，「テキストエンコーディング」というフォルダ内に「Japanese Encodings」やら「Unicode Encodings」といったさまざまなコード変換用のプラグインファイルが入っています(図2)。これらのファイルが入っていれば，TECがインストールされています。もしインストールされていない場合は，インストールを行ってください。MacOS 8以上であれば，CD-ROMに入っているはずですし，それよりも前のOSを使っている場合は，AppleのWebサイトへ行って，ダウンロードしてくればOKです。

TECがインストールされていてもいなくてもFB³からの呼び出しを行ううえで「Text Encoding Converter SDK」があるとより理解が深まるので，ぜひダウンロードしてきてください。これ以降の解説は，SDKに含まれるファイルが手元にあると仮定して進めていきます。ダウンロード先は以下のとおりです。

http://developer.apple.com/sdk/

なお，TECは実際には「Text Encoding Conversion Manager」によって管理されている「Text Encoding Converter」と「Unicode Converter」という2つのコンポーネントによって構成されています。今回はそのなかの「Text Encoding Converter」を用いて文字コード変換を行います。

FB³でのAPI呼び出しの定義

APIとはApplication Programming Interfaceの略で，簡単にいうと，とあるプログラム(OSを含む)が自分以外のプログラムからも利用できるように外部に公開している関数のことです。MacOSにおけるAPIはToolboxという名前で呼ばれています。Toolboxには，ウィンドウやボタン，スクロールバーなどさまざまなコントロールを描画するものを初めとしてたくさんの種類が用意されています。

さて，FBはToolboxを簡単に呼び出すことができますが，ToolboxはMacOSがバージョンアップするたびに追加・変更がなされていきます。ところが，FBIIまでは未定義のToolboxや外部のAPIを呼び出すためには，アセンブリ言語で記述を行わなくてはなりませんでした。そのため新しいライブラリの呼び出しを定義するためにはアセンブリ言語の知識が必要となってしまい，最新の技術を使うことの大きな障害になっていました。しかし，FB³ではその点について劇的な改善が行われました。FB³では，ライブラリ名と，呼び出しの書式を指定するだけで，簡単に新しいToolboxルーチンが定義できるようになったのです。たとえば，以下のような感じです。

```
TOOLBOX FN TECCountAvailableTextEncodings(ItemCount
 * numberEncodings)=OSStatus
```

なお，FB³では，標準で(つまり自分で定義しなくても)使用が可能なToolboxもすべて同様の方法で定義されています。これらは「FB Extensions:Compiler:Headers」というフォルダ内に内容別にいくつものファイルに分かれて定義されています。これらのファイルはプログラムをコンパイルする際に同時に取り込まれることで，これらのToolboxが使用できるようになっているのです。ですから，新たなToolboxを定義したいときは，これらのファイルでの定義方法を参考にしながら行うとよいでしょう。

FB³で文字コード変換(その1)～ヘッダファイルの変換

ではText Encoding ConverterをFB³から使ってみましょう。Text Encoding Converterで提供されている関数を使うためのヘッダが現在のところ用意されていません(Ver.3.3b3現在)ので，自分で用意しなければなりません。Appleは新しいライブラリを公開するときに，たいていC言語用とPascal用の2種類のヘッダファイルを一緒に公開します。Apple以外のどこかの誰かが作成，公開したライブラリなども，C言語のヘッダファイルがついてくるものがほとんどです。ですからそのヘッダファイルをもとにFB³用のインクルードファイルを作成すればよいわけです。では，C言語のヘッダからFB³のインクルードファイルを作る方法を考えてみましょう。変換はある程度規則的に行うことができます。いくつかの変換のポイントを以下に記します。

1．変数型の定義

C言語における「typedef」は，新しい変数型を定義するために使用するものです。たとえば以下のように使用します。

```
typedef OSType    TECPluginSignature;
```


図1　Text Encoding Converter ライブラリ

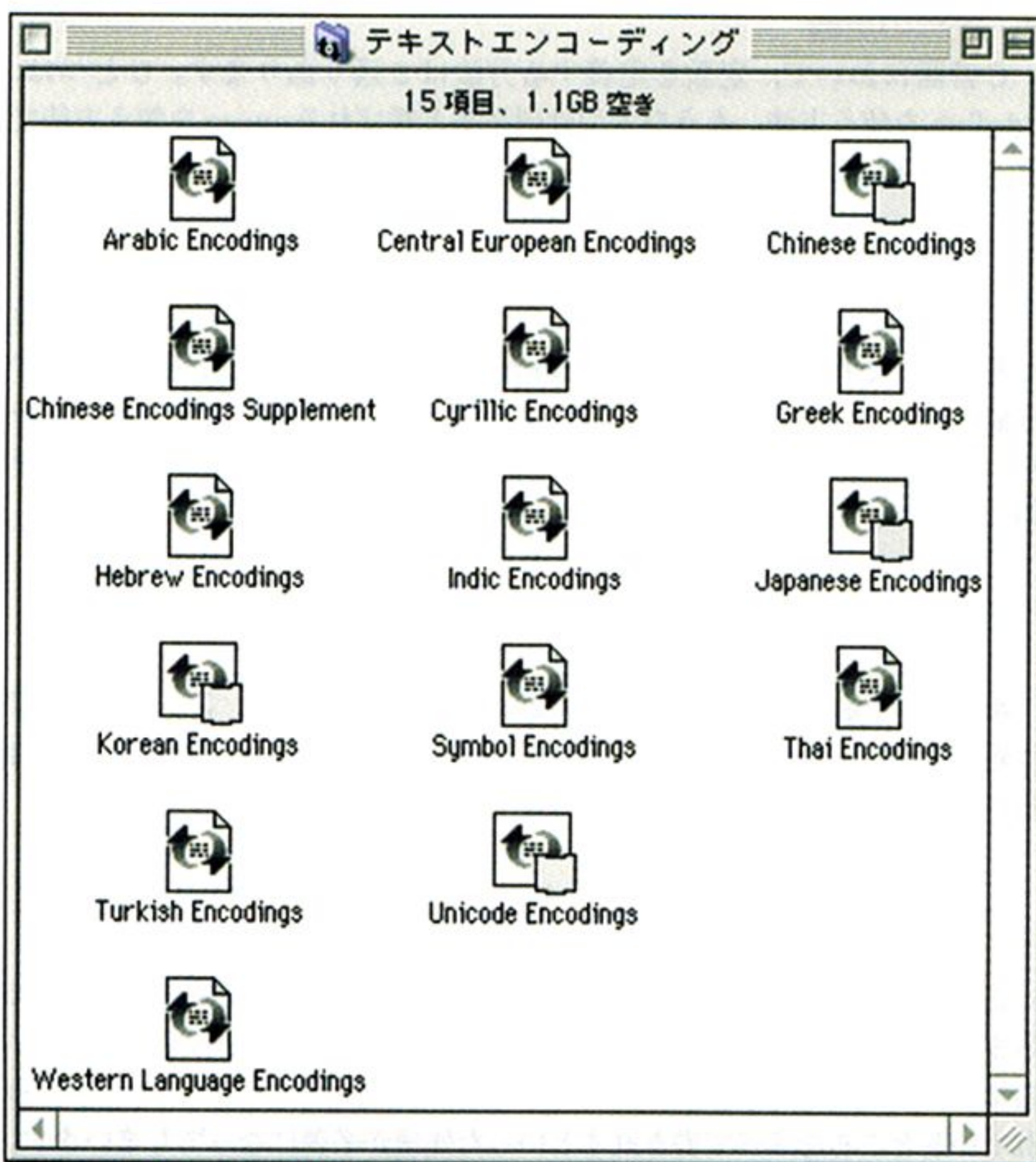



図2　各言語ごとのプラグインファイル

これは『「TECPluginSignature」という新しい変数型を「OSType」という変数型と同じ性質の変数型として定義する』という意味です。ちなみに「OSType」という変数型はC言語の標準の変数型ではありませんが，MacOSではよく使われている変数型で「MacType.h」という別のヘッダファイルで以下のように定義されています。

```
typedef unsigned long       FourCharCode;
typedef FourCharCode        OSType;
```

ここから「OSType」は「FourCharCode」と同じ変数型で「FourCharCode」は「unsigned long」と同じ変数型である，ということがわかります。「unsigned long」は4バイトの変数で符号なしの整数値を入れるものです。したがって，「TECPluginSignature」も「OSType」も「FourCharCode」も同じく4バイトの符号なし整数値を表す変数型ということになります。このように一度定義した変数型はさらに新しい変数型のひな型として使うことができます。

さて，これと同じ定義をFB³で行うには，#DEFINEという命令を使います。たとえば，最初の例だと以下のようになります。

```
#DEFINE TECPluginSignature AS OSType
```

「TECPluginSignature」と「OSType」の順序がC言語と逆なことに注意してください。ちなみに「FB³」においても「OSType」は標準の変数型ではありませんが，「Tlbx Standard.Incl」というファイルで以下のように定義されています。

```
#DEFINE OSType AS long
```

このようにFB³でもC言語と同じように一度定義した変数型はさらに新しい変数型のひな型として使うことができるわけです。「Tlbx Standard.Incl」は通常のコンパイル時には自動的に取り込まれるので，特に新たに定義する必要がありません。

なお，C言語にも#defineというものがありますが，これはFB³の#DEFINEとは異なる使い方をするものなので，注意が必要です。

2. 定数の定義

C言語において，定数を定義する方法は2通りあります。ひとつは，#defineを使う方法，もうひとつは列挙子と呼ばれるenumを使う方法です。このうち#defineを使った場合は，以下のようになっています。

```
#define  kMyPosition   100
```

と書けば，それ以降プログラム中でkMyPositionという文字列は100という値と見なされるようになるわけです。FB³では定数の前には「_（アンダーバー）」を置く決まりになっていますので，これをFB³に変換するには，以下のようにします。

```
_kMyPosition = 100
```

#defineで定義されているのが単なる定数だった場合は，これでよいのですが，C言語における#defineはもっと高機能なために，以下のような表現も可能です。

```
#define CHECK_CODE(c)   ((c) == 32 || (c) == 9 || (c) == 13 )
```

こういった場合はちょっとやっかいで，そのままFB³に変換することができません。このような#defineで簡単な変換式や条件式を作ってしまっている場合は，その部分をローカル関数として作り直すとか，その定義が使われているところをすべて書き直すといった処理が必要になってしまいます。

もうひとつの定数定義方法である列挙型を考えてみましょう。Appleが公開しているヘッダファイルなどでは，こちらのほうが圧倒的に多くなっています。たとえば，以下のような定義がされています。

```
enum {
    kTextEncodingFullName      = 0,
    kTextEncodingBaseName      = 1,
    kTextEncodingVariantName   = 2,
    kTextEncodingFormatName    = 3
};
```

これをFB³に変換すると，以下のようになります。FB³からはBEGIN ENUM～END ENUMという定義文が使えるようになったので，変換がとてもわかりやすくなりました。

```
BEGIN ENUM
  _kTextEncodingFullName  = 0
  _kTextEncodingBaseName  = 1
  _kTextEncodingVariantName = 2
  _kTextEncodingFormatName  = 3
END ENUM
```

また，TECのヘッダには以下のようなものもあります。

```
enum {
  kTECSignature                 = FOUR_CHAR_CODE('encv'),
  kTECUnicodePluginSignature    = FOUR_CHAR_CODE('puni'),
  kTECJapanesePluginSignature   = FOUR_CHAR_CODE('pjpn'),
  kTECChinesePluginSignature    = FOUR_CHAR_CODE('pzho'),
  kTECKoreanPluginSignature     = FOUR_CHAR_CODE('pkor')
};
```

FOUR_CHAR_CODEは，指定された4文字の文字列と同じ値を持つ4バイトの整数を返すために定義されているもので，FB³では以下のように書けば簡単に同じことができてしまいます。

```
BEGIN ENUM
  _kTECSignature                = _"encv"
  _kTECUnicodePluginSignature   = _"puni"
  _kTECJapanesePluginSignature= _"pjpn"
  _kTECChinesePluginSignature   = _"pzho"
  _kTECKoreanPluginSignature    = _"pkor"
END ENUM
```

3. レコード

C言語におけるstructは構造体を定義するための予約語です。これは以下のように使うことで，複数の変数の集合体を一度に扱うことができるようになるものです。

```
struct TECConversionInfo {
    TextEncoding        sourceEncoding;
    TextEncoding        destinationEncoding;
    UInt16              reserved1;
    UInt16              reserved2;
};
```

構造体はFB³ではレコードと呼ばれるものと同じものです。したがって，レコードを使えば同じ効果を得ることができます。上記のC言語の構造体をFB³のレコードに変換すると以下のようになります。typedefのところでも述べたように，C言語では変数型を先に書き，あとに実際の変数名がき

ますが，FBでは順番が逆になります。

```
BEGIN RECORD TECConversionInfo
  DIM sourceEncoding        AS TextEncoding
  DIM destinationEncoding   AS TextEncoding
  DIM reserved1             AS UInt16
  DIM reserved2             AS UInt16
END RECORD
```

この定義方法はFB³で新しく可能になったものです。この定義方法で定義されたレコードは「True Record（本物のレコード）」と呼ばれます。これに対して，FBIIでも行うことができた方法は「Pseudo-record（疑似レコード）」と呼ばれるものです。これは以下のようにDIM RECORDで定義するものです。

```
DIM RECORD TECConversionInfo
  DIM sourceEncoding        AS TextEncoding
  DIM destinationEncoding   AS TextEncoding
  DIM reserved1             AS UInt16
  DIM reserved2             AS UInt16
DIM END RECORD TECConversionInfoSize
```

この2つの定義方法の違いについては，ここではあまり深くは触れませんが，いままでのPseudo-recordでは，同じプログラム中の別々の場所で定義されたレコードであっても，同じフィールド名が使えなかったり，すでに存在する定数名と同じ名前のフィールドが使えなかったりと制約が多いものでした。C言語のヘッダファイルを変換したりした場合には，同じような名前のフィールドがたくさんある場合も多く，True Recordのほうが断然便利になっています。

4．API関数の宣言

続いては実際に呼び出すAPIの呼び出し規則を定義します。呼び出し規則とは，その関数を呼び出す際に必要な引数（パラメータ）の変数型と数，戻り値の有無と変数型を指定するものです。API関数はローカル関数と異なり，プログラム中に実際にそのプログラムが書かれているわけではないので，呼び出し規則をきちんと宣言してやらないと，コンパイラはその命令が正しいかどうかがわからないのです。C言語での宣言方法は以下のようになっています。

```
EXTERN_API( TextEncoding )
CreateTextEncoding   (TextEncodingBase   encodingBase,
        TextEncodingVariant  encodingVariant,
        TextEncodingFormat   encodingFormat);
```

これは，「CreateTextEncoding」という関数に関する定義文で，3つの引数（「TextEncodingBase型」のencodingBase，「TextEncodingVariant型」のencodingVariant，「TextEncodingFormat型」のencodingFormat）を必要として，戻り値が「TextEncoding型」であることを意味しています。この関数をFB³で宣言するには以下のようにします。この例では戻り値があるのでTOOLBOX FN文を使っています。戻り値がないものに関しては，TOOLBOX文を使用します。

```
TOOLBOX FN CreateTextEncoding(encodingBase AS
TextEncodingBase,
     encodingVariant AS TextEncodingVariant,
     encodingFormat AS TextEncodingFormat)
                                    = TextEncoding
```

しかし，FB³において，APIの宣言方法にはかなりの自由度が与えられており，これ以外にもいくつかの宣言方法が使えます。たとえば以下のようなものも，まったく同じような宣言となります。

(1) 引数を型だけしか記述しない方法

```
TOOLBOX FN CreateTextEncoding(TextEncodingBase,
     TextEncodingVariant,
     TextEncodingFormat) = TextEncoding
```

(2) 引数をC言語のように記述する方法

```
TOOLBOX FN CreateTextEncoding(TextEncodingBase
                                        encodingBase,
     TextEncodingVariant encodingVariant,
     TextEncodingFormat encodingFormat) = TextEncoding
```

FB³におけるAPIの宣言方法はC言語からの変換をかなり強く意識しており，C言語の宣言がほとんどそのまま使えるようになっているのです。このおかげで，ヘッダファイルの変換の効率は劇的に上昇します。これらの宣言方法については，FBの発売元であるSTAZのWebサイトで公開されているFAQ（#10: How do I add new toolbox calls?）に詳しい説明がありますので，参照してください。

http://www.stazsoftware.com/fb_faq10.html

5．ライブラリの指定

新しいAPIを定義するにあたって，忘れてはいけないのは，ライブラリの指定です。MacOSではさまざまなAPI（Toolbox）がさまざまなライブラリに分類されて存在しています。「機能拡張」フォルダを覗くと「ライブラリ」という種類のファイルがいくつも入っています。これらのファイルはAPIを提供する共有ライブラリファイルです。ですから新しいAPIを呼び出す際には「そのAPIがどこにあるのか」ということをきちんと指定してやらなければいけないのです。もちろん新しい共有ライブラリを作ってFBから呼び出すこともできます。

ライブラリの呼び出しには「LIBRARY」という命令を使います。今回の場合では，「TextCommon.h」「TextEncodingConverter.h」「UnicodeConverter.h」の3つのヘッダはそれぞれ「TextCommon」「TextEncodingConverter」「UnicodeConverter」というライブラリ名を指定します。LIBRARY文は一度指定すると，もう一度別のLIBRARY文を指定するまで有効になります。ですから，各ファイルの最後では，「LIBRARY」をライブラリ名なしで指定して，初期状態に戻しています。

```
LIBRARY "TextCommon"
:
:  (APIの宣言)
:
LIBRARY
```

なお，ライブラリの指定はAPI関数の宣言のみに影響しますので，レコードや新しい型の宣言などはライブラリを指定する前でもかまいません。

さてLIBRARYという命令に対応するC言語の命令は存在しません。C言語では，ライブラリの指定の仕方がまったく異なるためです。しかも，指定するライブラリの名前がヘッダファイルには含まれていないので，なんらかの方法で調べる必要があります。ライブラリ名は，「TextCommon」なんてファイルがないことからもわかるように，ライブラリファイルのファイル名と同じかと思うと，そういうわけでもありません。

FB³に含まれているヘルプファイルのなかに「Mac Libraries」というファイルがあり，そのなかに現在わかっているライブラリの名前の一覧が書かれています。たいていのファイルはここで見つけることができますが，なかには使いたいのに見当たらないものもあります。たとえば，「OpenTransport」関係のライブラリの名称がありません。

以前，OpenTransportが使いたくていろいろ調べたことがあるのですが，初期化するToolboxである「InitOpenTransport」という命令の含まれるラ

イブラリの名前が見つかりませんでした。STAZ（FB^{3}の開発元）に直接問い合わせたところ「知ってるライブラリはすべてヘルプファイルに書いた。あとはわからない」といわれ，メーリングリストでも誰もわからず，大変困ったのでした。しかも，最終的にわかったことは，この命令は外部に公開されていないので，どうやらFB^{3}からは使えないらしいということでした。このように，ライブラリによってはうまく使うことができなかったりしますので，新しいライブラリを使いたいと思ったとき，いちばん苦労するのがLIBRARYの指定かもしれません。

そのほかにも特殊な指定方法をするものはあります。「InternetConfig」を使うときは「ICAp;InternetConfigLib」という風に指定しなければなりません。なんでこういう名前なのかはよくわかりません。指定の仕方が全然わからなかったので，ライブラリファイルを片っ端からバイナリエディタで開いてなかにある文字列を調べていったところ発見できたものだからです。かなり乱暴な方法ですが，たまにはこういった探し方も有効なことがあります。

FB^{3}のサンプルファイルに「Shared Library Listing」というサンプルがあります。これを使うと，ライブラリの名前からそのライブラリに入っている命令の一覧を知ることができます。ライブラリの名前が正しいかどうかを調べるときにも有効です。

ヘッダファイルの変換を行ううえでもっとも重要なポイントはだいたいこんな感じです。それでは実際にヘッダファイルを変換してみます。SDKのフォルダのなかに「Headers」というサブフォルダがあり，そのなかに4つのヘッダファイルが入っています。このうち「TextEncodingPlugin.h」はTECに扱える文字コードを追加するプラグインを作成するためのヘッダファイルですので，今回は必要ありません。それ以外にも3つのヘッダファイルがTECを使うためのヘッダファイルです。TECはText Encoding ConverterとUnicode Converterという2つのコンポーネントから構成されていますが，今回はUnicode Converterは使用しないので，実際には「UnicodeConverter.h」を変換する必要はありません。

すでに変換したものは，CD-ROMや筆者のWebサイトから手に入れられますので，参考にしてみてください。それぞれのファイルの名前は，「〜.h」ではなく，FB^{3}のインクルードファイルの一般的な拡張子「〜.Incl」にしてあります。

FB^{3}で文字コード変換(その2) 〜実際の変換

では，先ほど作った3つのインクルードファイルを，プロジェクトに追加します（図3）。これでもう，TECを使うための準備は完了です。あとは，実際に呼び出してみるだけです。リスト1に実際に変換を行うためのプログラムを示します。文字コードはTextEncodingという4バイトの変数を使って保持しています。この変数のために文字コードを指定する方法はいくつかあるのですが，今回は「InternetName」を使って指定してみます。これは文字コードを文字列で指定できますので，直感的にいちばんわかりやすい方法だと思います。

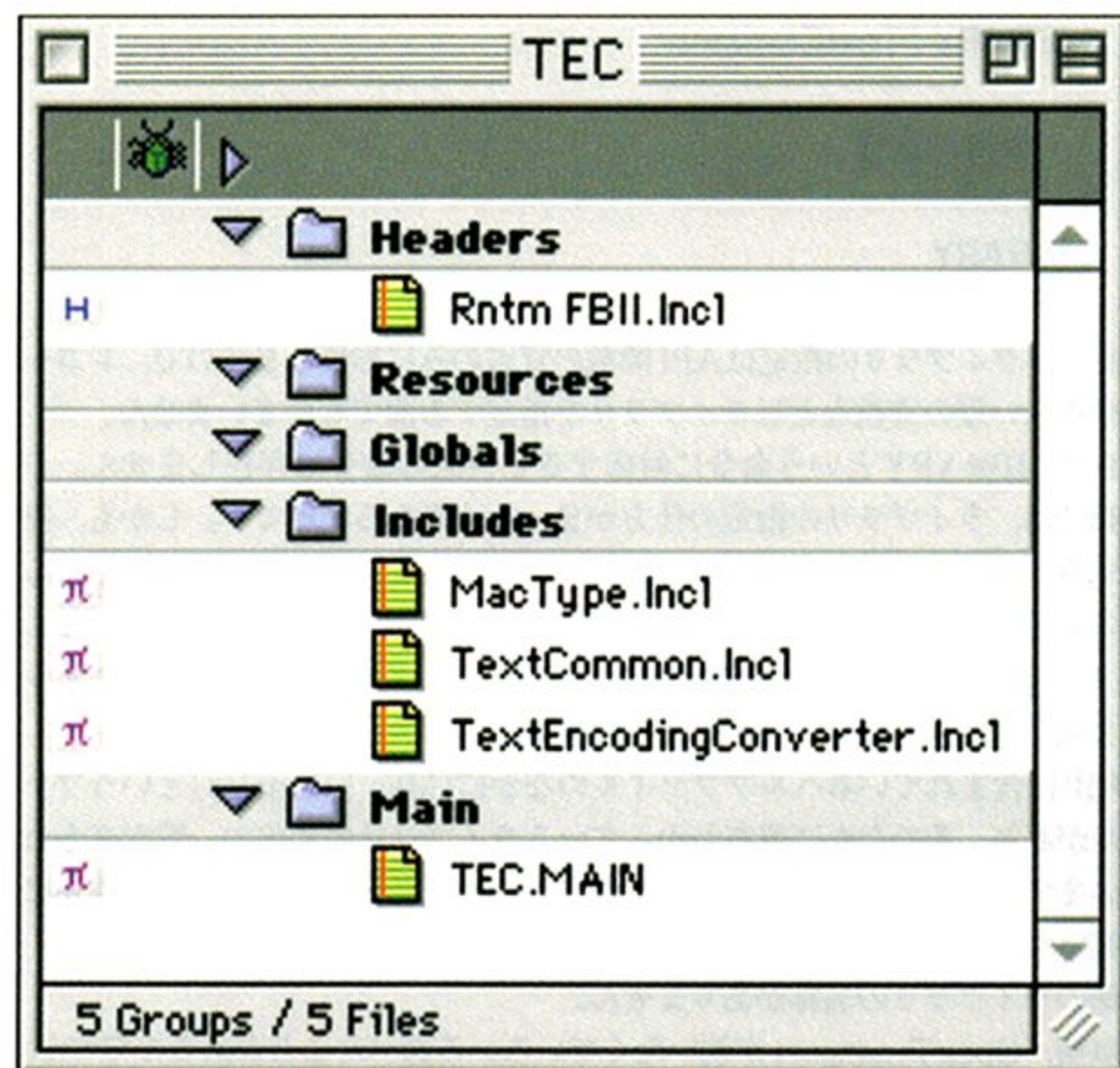



図3　プロジェクトにTEC関連のファイルを追加

TECGetTextEncodingFromInternetNameというとても長い名前のAPIを利用して，文字コードの指定を行います。今回のサンプルでは「SHIFT-JIS」から「UTF-8」に変換を行っているので，inputEncoding（変換元の文字コード）とoutputEncoding（変換先の文字コード）にそれぞれの文字コードを指定しています。ここで指定している「Shift_JIS」や「UTF-8」などは，IANA（Internet Assigned Numbers Authority）によって管理されている名称で，HTMLでのキャラクターセットの指定などでも使われている非常にポピュラーなものです。

TECで使用することができるコードはSDKに添付されているドキュメントに一覧表になっていますが，日本語に関係するものを表1に示します。複数の名前で指定できる文字コードも存在します。たとえばShift-JISは，「Shift_JIS」のほかにも「x-sjis」や「x-shift-jis」という名前で指定できるのです。

文字コードの指定ができたら，続いて，変換を行うためのTECオブジェクト本体を生成します。TECObjectRefという参照変数を用意して，TECCreateConverterを呼び出すことで，TECオブジェクトを生成することができます。このオブジェクトこそが変換を行うための「マシーン」ともいえる存在で，このオブジェクトに文字列を放り込むことで，文字列変換を行うことができるわけです。そのための関数がTECConvertTextです。この関数にTECオブジェクト，変換元文字列へのポインタと長さ，変換した文字列を格納するバッファのポインタと長さを指定してやれば変換が行われるわけです。なお，inputactとoutputactは実際に変換した変換元文字列の長さ，変換後の実際の長さが返る変数です。

そして最後に，TECDisposeConverterで使い終わったTECオブジェクトを破棄すれば終了です。

なお，このサンプルではリストの流れをつかみやすくするために，エラー処理をまったくしていませんが，実際に使用する際にはエラーをチェックしたほうがいいでしょう。このサンプルの場合，エラーはすべてxErrという

リスト1　InternetNameを使った文字コード変換

```
DIM xErr AS OSStatus
DIM inputEncoding AS TextEncoding
DIM outputEncoding AS TextEncoding
DIM converter AS TECObjectRef
DIM inputact AS ByteCount
DIM outputact AS ByteCount
DIM a$, b$
DIM d AS BYTE

WINDOW #1,"test"

a$ = "テスト文字列"

xErr = FN TECGetTextEncodingFromInternetName(@inputEncoding,"Shift_JIS")
xErr = FN TECGetTextEncodingFromInternetName(@outputEncoding,"UTF-8")
xErr = FN TECCreateConverter (@converter, inputEncoding, outputEncoding)
xErr  =  FN  TECConvertText  (converter,  @a$+1,  |@a$|,  @inputact,@b$+1,255,
@outputact)
xErr = FN TECDisposeConverter(converter)

d = outputact
| @b$,d
PRINT b$
STOP
```

表1　Text Encoding Converterで使うことができる日本語に関連するInternetName

文字コード	Internet Name
Unicode 2.0 (16 bit)	UTF-16
Unicode 2.0 UTF-8	UTF-8
Unicode 2.0 UTF-7	UTF-7
ASCII	US-ASCII
ISO 8859-1 (Latin-1)	ISO-8859-1, latin1
EUC-JP	EUC-JP, X-EUC-JP
ISO 2022-JP ("JIS")	ISO-2022-JP
Shift-JIS	Shift_JIS, x-sjis, x-shift-jis

変数に入ります。

さて，最後にFB³における文字列の扱い方を説明しておきます。FBでは文字列はパスカル文字列という形式で保存されます。これは，最大255文字を格納できる方式で，いちばん先頭に文字列の長さを格納しています。したがって，必要とするメモリ量は文字列の最大長に1を加えた長さになります。今回のような文字列長と実際の文字列が開始されるメモリ位置を渡さなければいけない場合は，リスト2のようにすればよいわけです。

FB³で文字コード変換(その3)～文字コードの判定

さあ，これで文字コードの変換が行えるようになりましたが，変換前の文字コードを直接指定しているので，このままでは変換元の文字コードがすでにわかっている場合しか対応できません。しかし文字コードがなんであるかはわからない場合もあります。そのためにTECでは，文字コードの判定を行う機能を提供しています。

文字コードをチェックするための関数をリスト3に示します。文字コードのチェックは結構複雑で，それに伴ってリストも長くなってしまいました。例によってエラーチェックは甘アマなので，必要に応じてエラー強化を行ってください。

文字コードの判定を行うには，Sniffer［探知機］オブジェクトというものを使います。Snifferとは，ここの文字コードごとに提供されている与えられた文字列がその文字コードのコード体系に適合しているかをチェックするシステムで，TECで使うことのできるすべての文字コードで提供されているわけではありません。そこで，FN TECGetAvailableSniffersという関数を用いてSnifferを使用可能な文字コード(TextEncoding型)を取得します。そしてFN TECCreateSnifferを使ってSnifferを生成すれば，FN TECSniffTextEncodingで文字コードのチェックが行えます。

しかし，文字コードのコード領域はもちろん重複しています。したがってこのままでは，日本語の文字列なのにコードが別の言語の文字コードの範囲にも矛盾しなかったために，全然異なった言語の文字コードだと判定される可能性もあります。そこで，その危険を排除するために，FN TECGetWebTextEncodingsという関数を使って文字コードを取得しておきます。これは，領域コード(日本の場合は14)を指定すると，その領域(地域)で使われている文字コードのみを返してくれる関数です。この関数で得られた文字コードのうち，Snifferが使えるものだけを調べて，それらの文字コードのみで構成されるSnifferを生成しているわけです。これによって，ほぼ正しい(と期待される)文字コードを判定することができます。なお，リスト3では領域コードを14(日本)で決め打ちしているので，Shift_JIS，EUC-JP，ISO-2022-JPの3種類しか判定することができません。領域コードの一覧については，以下のURLなどを参照してください。

http://developer.apple.com/techpubs/macos8/TextIntlSvcs/ATSUI/ATSUI_ref/ATSUI-202.html

終わりに

さて，TextEncodingConverterを題材にして，FB³におけるAPI呼び出しの方法について簡単に解説してみましたが，いかがだったでしょうか。本当は，文字コード変換のアルゴリズムについて書くつもりだったのですが，いろいろ実験しているうちに，こういうカタチになりました。こういったヘッダファイルもそうですが，FB³では，レコードや#DEFINEなどの実装により，C言語からの変換がかなり容易になっています。C言語はさすがにもっともポピュラーな言語だけあって，さまざまなプログラムが公開されています。変換の規則さえわかってしまえば，FB³で動くように変換するのは簡単ですから，これを機会に，C言語で書かれたプログラムをFB³に変換してみるといいかもしれません。

リスト2　文字列の長さとデータ開始位置の指定の仕方

```
DIM strings    AS STR255
DIM length     AS INTEGER
DIM strpointer AS POINTER

strings    = "Test String"
length     = |@strings|          // 文字列長を取得
strpointer = @strings + 1        // 実際の文字列の開始位置
```

リスト3　文字コードを判定する

```
CLEAR LOCAL
LOCAL FN FindBestEncoding&(string AS STR255)
 DIM err                    AS OSStatus
 DIM maxRegionEncodings     AS ItemCount
 DIM actualRegionEncodings  AS ItemCount
 DIM maxSnifferEncodings    AS ItemCount
 DIM actualSnifferEncodings AS ItemCount
 DIM regionEncodings        AS POINTER TO TextEncoding
 DIM snifferEncodings       AS POINTER TO TextEncoding
 DIM sniffer                AS TECSnifferObjectRef
 DIM errors                 AS POINTER TO ItemCount
 DIM features               AS POINTER TO ItemCount
 DIM loop                   AS INTEGER
 DIM loop2                  AS INTEGER
 DIM locale                 AS INTEGER
 DIM bestEncoding           AS TextEncoding
 DIM found                  AS BOOLEAN

 locale = 14  //verJapan

 err = FN TECCountWebTextEncodings(locale, @maxRegionEncodings)
 regionEncodings = 0
 regionEncodings = FN NewPtrClear (SIZEOF(TextEncoding) * maxRegionEncodings)
 LONG IF regionEncodings>0
  err = FN TECGetWebTextEncodings (locale, regionEncodings,maxRegionEncodings,
@actualRegionEncodings)
  err = FN TECCountAvailableSniffers (@maxSnifferEncodings)
  snifferEncodings=0
  snifferEncodings = FN NewPtrClear (SIZEOF(TextEncoding) * maxSnifferEncodings)
  LONG IF snifferEncodings>0
    err = FN TECGetAvailableSniffers (snifferEncodings, maxSnifferEncodings,
@actualSnifferEncodings)
    loop = 0
    WHILE loop < actualRegionEncodings
      found = _false
      FOR loop2 = 0 TO actualSnifferEncodings-1
           LONG     IF     [regionEncodings+(loop*SIZEOF(TextEncoding))]     =
[snifferEncodings+(SIZEOF(TextEncoding)*loop2)]
           found = _true
           EXIT FOR
       END IF
      NEXT
      LONG IF found>0
       INC(loop)
      XELSE
       actualRegionEncodings = actualRegionEncodings - 1
       & (regionEncodings+SIZEOF(TextEncoding)*loop) ,
[regionEncodings+(SIZEOF(TextEncoding)*actualRegionEncodings)]
      END IF
    WEND

    LONG IF actualRegionEncodings = 0
      EXIT FN
    END IF
    err = FN TECCreateSniffer (@sniffer, regionEncodings, actualRegionEncodings)
    errors=0
    errors = FN NewPtrClear (SIZEOF(ItemCount) * actualRegionEncodings)

    LONG IF errors>0
     features=0
     features = FN NewPtrClear (SIZEOF(ItemCount) * actualRegionEncodings)
     LONG IF features>0
           err  =  FN  TECSniffTextEncoding  (sniffer,  @string+1,  |@string|,
regionEncodings, actualRegionEncodings, errors, actualRegionEncodings, features,
actualRegionEncodings)
      bestEncoding = [regionEncodings]
      CALL DisposePtr (features)
     END IF

     CALL DisposePtr (errors)
    END IF
    LONG if sniffer>0
     CALL TECDisposeSniffer(sniffer)
    END IF
    CALL DisposePtr (snifferEncodings)

   END IF
   CALL DisposePtr (regionEncodings)
  END IF
END FN=bestEncoding
```

Javaゲーム制作講座(1)
スプライトマネージャの使い方

菊地 功 Kikuchi Isawo

iモードに搭載されたこともあって，Java人気も再燃しそうな今日この頃。Javaを使って手軽にスプライト型ゲームを作成するためのライブラリがスプライトマネージャです。キャラクターの表示と管理の手間を大幅に軽減してくれます。

一時期は寝ても覚めてもJavaと騒がれていたのが，最近ではすっかり落ち着いて，あまり耳にすることもなくなった。インターネットを漂っていても，あまり見かけることもない。というか，確かにJavaはネットワークに対応した優れた言語ではあるが，C++をベースにした本格的な言語であるので，そもそも言語すら知らない一般人がそんなものに興味を示していたこと自体が特異なことだったともいえる。そういえば，C++も出た当時は本格オブジェクト指向言語として，かなり露出していたような覚えがある。つまり，Javaも素人がわけもわからずもてはやす時代を過ぎ，着実に根づいてきているということだろう。余談だが，JavaとJavaScriptはまったくの別物である。サブセットとかそういうレベルでもなく，言語仕様自体が別次元といってよい。というか，JavaScriptはあくまでスクリプトであり，言語とはいえないかもしれない。「これがあのJavaかぁ」とかいいながら，ロールオーバーボタンを作って喜んでいる君，それは大きな間違いだぞ。なお，噂によると，年末辺りからiモードでJavaが動くようになるそうだが，メモリ容量(と画面解像度)を考えると，当初の1～2年はほとんど期待できなそうだ。

などと偉そうなことを抜かしながら，実は筆者もJavaは大昔に触ったきりだったりする。言語自体が新しいものだから，大昔といってもたかだか3～4年前のことだが，この業界の進歩を考えると十分すぎる大昔だ。確か筆者はJDK (Java Development Kit) 1.0.2を使っていて，1.1のβが出ていて，1.2の噂が囁かれていた頃だったと記憶している。JIT (Just In Time) コンパイラも出るか出ないかといった段階で，重たすぎてろくなものは作れなかった頃だ(編注：JIT自体は出てたはずだが)。

それが最近ちょっとJavaの必要があって調べてみると，いつのまにかJava2になっている。おお，なにか知らないうちに進んでんじゃねーか，と思ってさらに調べてみると，1.2をJava2という名称に変更したとのこと。……進んでねーじゃねーか。ま，裏を返せば完成の域に達したってことか。言語がそんなにぽんぽんバージョンアップしても，たまったもんじゃないしな。ちなみにJDKという表記は廃止になり，開発キットはJava2 SDKと呼ぶことになったらしい。

そんなわけで，今号のOh!XはJava普及号と筆者が勝手に銘打って，Javaの普及に努めたい。巷ではJava2DやJava3D (Java2に被せる拡張クラスライブラリみたいなもの？)といったものが出回っているようだが，ここではそういったものではなく(だって触ったことないんだもん)，筆者が昔作ったスプライトマネージャでゲームを作ることを最終目標とする。Java自体が透過GIFをサポートするので，単にGIFを表示するだけでもスプライトっぽい表示は可能なのだが，もっとシステマティックにマネジメントするライブラリである。

また，ムツカシイ話や細かい言語仕様は省略する。そういうのは教科書を読んで詰め込むよりも，実戦で慣れるのがいちばんである。とにかく動くゲームができればいい(多くの人がそうだと思う)ということを大前提にするので，それ以上のことを知りたい人は，別途参考書でも用意してほしい。逆にC++を知っている人は，先ほども述べたようにJavaはC++がベースになっている(念のためにいっておくと，互換性はない)ので，C++で身につけた知識が役に立つだろう。ただし，当然ながら違う点も少なからずあるので，C++を知っているだけに混乱してしまいそうな点もあることに注意してほしい。

まずC言語で8割の人がつまずくといわれているポインタと構造体がない(配列はある)。挫折した人にとっては嬉しいかもしれないが，使いこなしていた人にとっては非常に不便を感じる。しかし，C++で8割の人がつまずくといわれているクラスは大前提として存在する。そういう意味では，いきなり高いハードルが目の前にそびえている。Javaではクラスなしにプログラムは組めない(というより，クラスしかない。グローバルは存在しない)ので，ここまでクラスを避けて通っていた人は覚悟を決めてほしい。オブジェクト指向言語にクラスはつきものだ。とはいえ，何度もいうようにちょっとゲームを作りたいだけなので，今回はクラスで難しいことをするつもりはない。クラスだけをいうなら，JavaScriptもクラスの概念(ドット"."のついたアレね)を採用したスクリプトなので，そちらに慣れている人ならば違和感なくすんなり触れるかもしれない(厳密には，派生までできて初めてクラスといえるのかもしれないが)。

また，確かにポインタはないのだが，たとえばクラスFooを宣言するため，

```
Foo foo;
```

としても，Fooのインスタンスは確保されない。インスタンスを確保するためには，

```
Foo foo = new Foo;
```

と記述する必要がある。つまりクラスを宣言する変数fooは，Cでいうところのポインタ宣言となる。これがもし単純な型変数であればもちろん，

```
int foo;
```

で整数型の変数が確保される。なにか素直に納得できない言語仕様ではあるが，ポインタを使わせたくない苦肉の策なのだろう。

また，newがあるのにdeleteがないなど，上級者であるほど気持ち悪くなる仕様もある。メモリは不要になったときにシステムが自動的に解放するということだが(実際には，メモリが足りなくなったときに不要なメモリをサーチして解放していると思われる)，明示的に解放させてくれてもいいと思うのだが。

下準備

Javaの統合開発環境は各社から発売されているが，本当に必要なものはJava2 SDKとテキストエディタだけである(余談だが，MicrosoftのVisual J++は原稿執筆時点でJDK1.1にしか対応していない)。Java2 SDKはhttp://java.sun.com/からダウンロードできるが(原稿執筆時点の最新版はJava2 SDK v1.3)，SDKが約30MB，ドキュメントが約20MBあり，決して小さくはない。JavaSDKはver.1.0になる前は自由に配布できたのだが，1.0以降は正式商品ということで雑誌などでの配布はできなくなった。頑張ってダウンロードするか，Java2対応の開発ツールに入っていることがあるので，そういったものを導入して対処してほしい。ピュアなSDKとテキストエディタだけでプログラムすることを前提に話を進める。JDKをインストールしたら，まずはインストールしたフォルダの中のbinフォルダにパスを通しておこう。

次にCD-ROMに収録したspman040.lzhを，どこか適当なフォルダに解凍してほしい。ファイルがいくつか作られるが，このうちマネージャとして

必要なファイルはSpriteLib.jarだけだ。このファイルは基本的にこれから作成するJavaのアプレット（Webに貼り付けられるJavaもモジュールをアプレットという）と同じフォルダに置く必要がある。なお，このアーカイブにはサンプルも2つ収録してあるので，先に見ておけばどういったものが作れるかイメージが湧くだろう（図1.2）。

キャラクターの表示

とりあえずキャラクターを表示してみたのがリスト1である。順に説明していこう。

まず1，2行目のimportは，Cでいうところのincludeに相当する。これから記述するプログラムの中で，必要となるクラスの定義を参照するためのもので，パッケージと呼ばれる。java.applet.AppletではAppletクラスが定義されており，アプレットを作るには必ず必要となる。java.awt.＊というのは，awtに含まれるすべてのパッケージを参照しなさいってことだ（AWTはAbstract Window Toolkitの略）。画像やフォントなど，さまざまなものを扱うためのパッケージが含まれているが，1つひとつimportするのも面倒なので，まとめてimportしてしまうのが手っ取り早い。

4行目からがクラスの宣言である。もしjavaソースのファイル名をjshooting.javaとしたなら（拡張子はjavaでなければならない），ここのクラス名もjshootingとしなければならない。大文字小文字も正しくだ。コンパイルの方法はのちほど説明するが，コンパイルするとjshooting.classというファイルが作成され，それをHTMLから指定すると，その中の同名のクラスを検索し，インスタンスの作成・実行をシステムが行う仕組みになっている。後ろにくっついているextends Appletというのは，Appletクラスをサブクラス化するという意味で，アプレットで最初に呼ばれるクラスには必ずつける必要がある。importからここまではお約束と思って覚えてしまえばよいだろう。

5～7行目はクラス変数の宣言なのでとりあえず置いといて，8行目のinit()メソッド（関数）である。これはAppletクラスで宣言されているもののオーバーライドである。アプレットもWindowsと同じく，基本はイベントドリブンであり，init()はアプレットが実行されてまず最初に呼ばれるメソッドだ。ここで必要な初期化を行う。まずスプライトマネージャに含まれるSpriteControlクラスの初期化だ。スプライトマネージャは，初期化時にスプライトパターンとスプライトプレーンの数を指定する必要があるが，理論的にはそれぞれint型の最大値（約21億個）まで確保できる。次の引数はスプライトを表示する領域の幅と高さ。この場合はアプレットの大きさそのものとなる。

最後はコンポーネントと呼ばれるものを指定するが，ここは深く考えずにthisとしておいてほしい。C++のthisポインタと同じもので，今自分がいるクラスのポインタということだが，スプライトが表示される先がこのクラスだと思っておけばよい。次はMediaTrackerクラスの初期化だ。javaアプレットは，オンライン上のサーバ内にあるわけで，画像やサウンドなどのデータ類ももちろんサーバにある。そのため，そういったデータをロードしようとした場合，線によっては時間がかかる場合がある。その間ずっとプログラムが停止してしまわないように，データのロードは必要になったときに，バックグラウンドで行われる。そのロードを促し，状態を調べるのがMediaTrackerクラスである。画像などを扱う場合には必須だ。そのコンストラクタも引数としてコンポーネントを取るが，ここもthisとしておけばよい。

11行目は画像のロードだ。getImage()の第1引数はURLを示す。getDocumentBase()は，アプレットがロードされたURLを返すので，アプレットがあるディレクトリの中の，imgというディレクトリにあるback.gifというファイルをロードしなさい，ということになる。それを先ほどのMediaTrackerのリストに加える。第2引数はIDを示し，このID単位

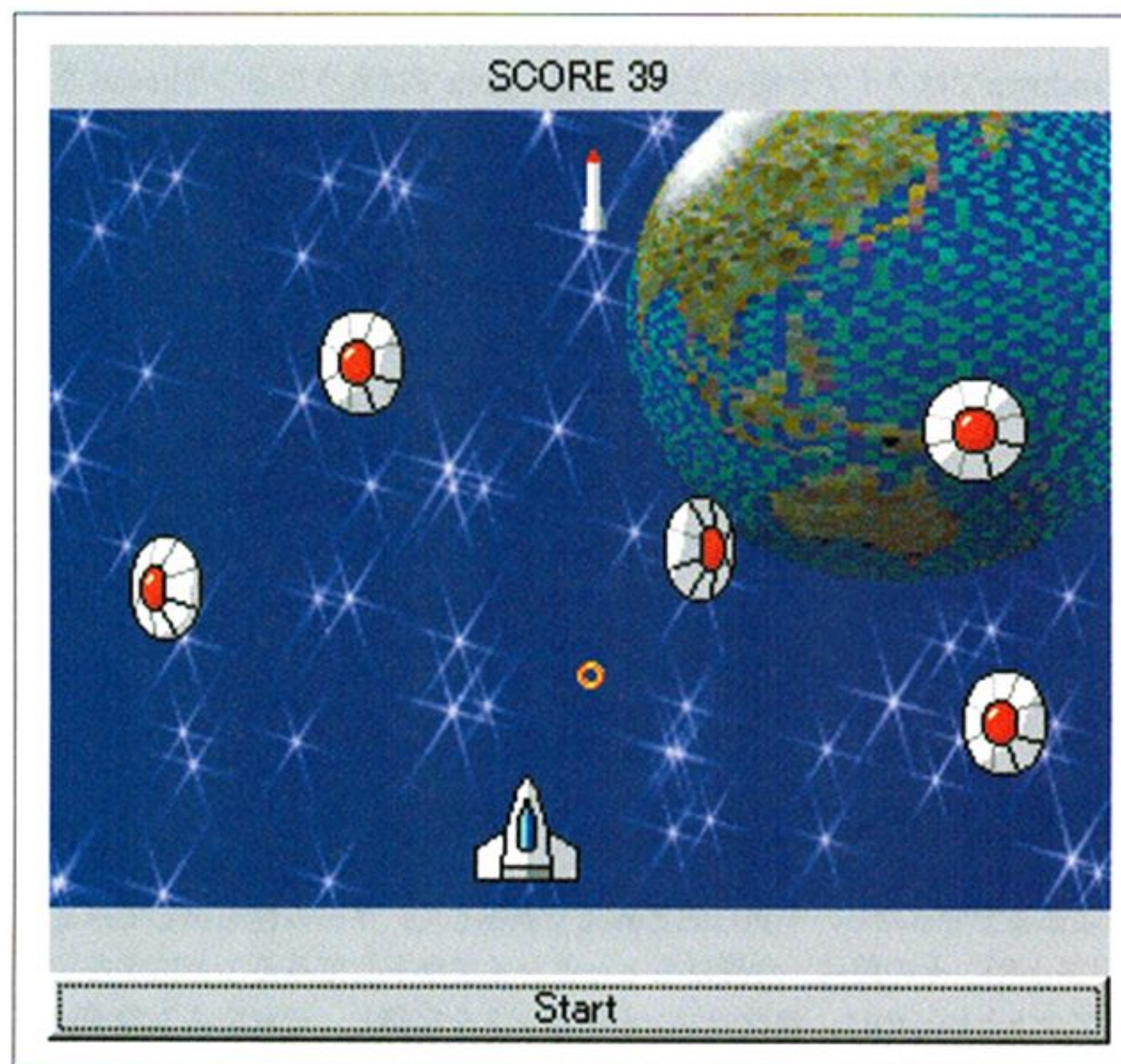



図1 サンプルゲームShootSoucer

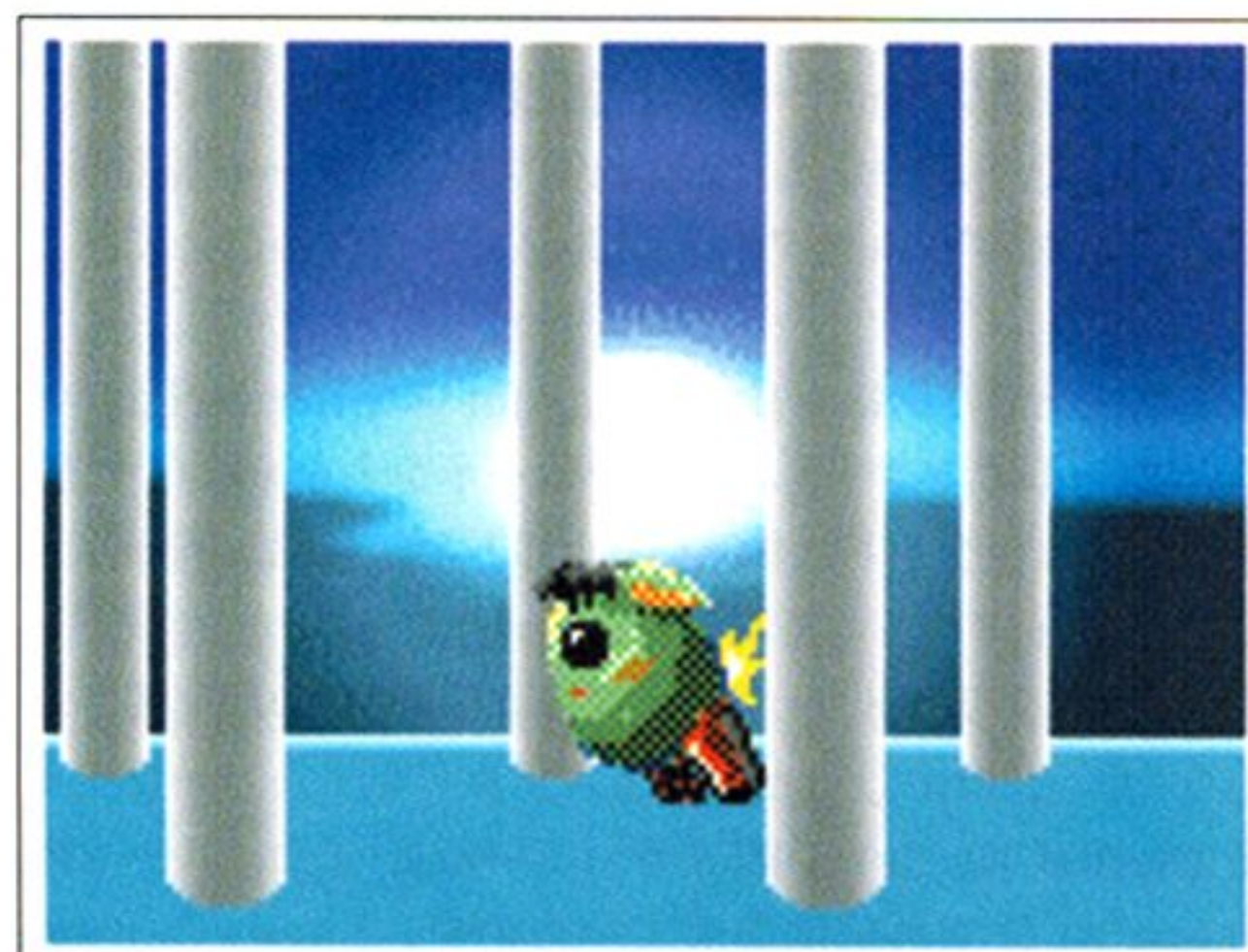



図2 バックグラウンドサンプルscroll

リスト1

```
[1]import java.applet.Applet;
[2]import java.awt.*;
[3]
[4]public class jshooting extends Applet {
[5] static final int WIDTH = 240, HEIGHT = 320; // 背景サイズ
[6] SpriteControl sc; // スプライトコントロール
[7] MediaTracker mt;
[8] public void init(){
[9]   sc = new SpriteControl( 1, 1, WIDTH, HEIGHT, this );
[10]    mt = new MediaTracker( this );
[11]    Image image = getImage( getDocumentBase(), "img/back.gif" );  // 背景
[12]    mt.addImage( image, 0 );
[13]    sc.SetBGImage( image );
[14]    image = getImage( getDocumentBase(), "img/myship.gif" );  // 自機
[15]    mt.addImage( image, 0 );
[16]    sc.Define( 0, image );
[17]    sc.Set( 0, 0 );
[18]    sc.Move( 0, (240-32)/2, (320-32)/2 );
[19]    sc.Show();
[20]    try {
[21]      mt.waitForID( 0 );
[22]    } catch( InterruptedException e ){
[23]      return;
[24]    }
[25]  }
[26]  public void update( Graphics g ){
[27]    paint( g );
[28]  }
[29]  public void paint( Graphics g ){
[30]    if( mt.checkID( 0 ) ){
[31]      sc.Display( g, this );
[32]    } else {
[33]      g.drawString( "Loading...", 0, 12 );
[34]    }
[35]  }
[36]}
```

でメディアの制御を行う。13行目は，そのback.gifをスプライトコントロールの背景とするメソッドだ。

14，15行目も同じだ。16行目以降では，今度はスプライトパターンとして，myship.gifをパターン0に登録，そのパターン0をプレーン0にセット，プレーン0の座標を設定している。

19行目はスプライトを表示状態にするメソッドで，引数がない場合はすべてのプレーンを表示状態にする（あくまでも表示状態の変更であり，画面に表示するためのメソッドではない）。20～24行は，先ほどMediaTrackerに登録した画像の読み込み待ちだ。try catchで例外処理をトラップしているが，深く気にせずに，これもこういうものだと思っておいてもらいたい。

さて，update ()とpaint ()メソッドだが，これもAppletクラスのオーバーライドだ。ともに画面を描画するときに呼ばれるメソッドだが，update ()はユーザーがrepaint ()といった更新メソッドを呼んだときに呼ばれるもの，paint ()はほかのウィンドウなどで隠されていたアプレットが，再び表示されるときに呼ばれるものらしい。たいていの場合はやることは同じなので，update ()からpaint ()を呼んでおけば問題ない。

そのpaint ()メソッドだが，MediaTrackerで画像のロードチェックを行って，振り分けている。checkID (0)は，先ほど追加したID0のイメージが，すべてロードされていればtrueを，まだロード中ならばfalseを返す(init ()メソッドの最後でロード待ちをしているので，実際にはここはtrueにしかなりえない)。もしロードが完了していれば，SpriteControl::Display ()でスプライトを描画し，まだなら"Loading..."の文字を表示する。Display ()の第2引数はイメージオブザーバーを指定するが，これもthisでOKだ。

ソースが書けたら，コンパイルを行う。SDKに付属のjavac.exeというコンパイラを使うのだが，これはコマンドライン型のプログラムなので，DOS窓を開くなり，バッチファイルを作るなりしよう。以下のように実行する。

```
javac -classpath SpriteLib.jar jshooting.java
```

-classpathオプションは，後ろに続くSpriteLib.jarから参照するクラスを探すことを意味している。この場合はSpriteControlクラスだ。エラーがなければ，同じフォルダにjshooting.classというファイルができているはずだ。そこで，今度はHTMLで次のように記述する。

```
<applet code="jshooting" archive="SpriteLib.jar"
width=240 height=320>
</applet>
```

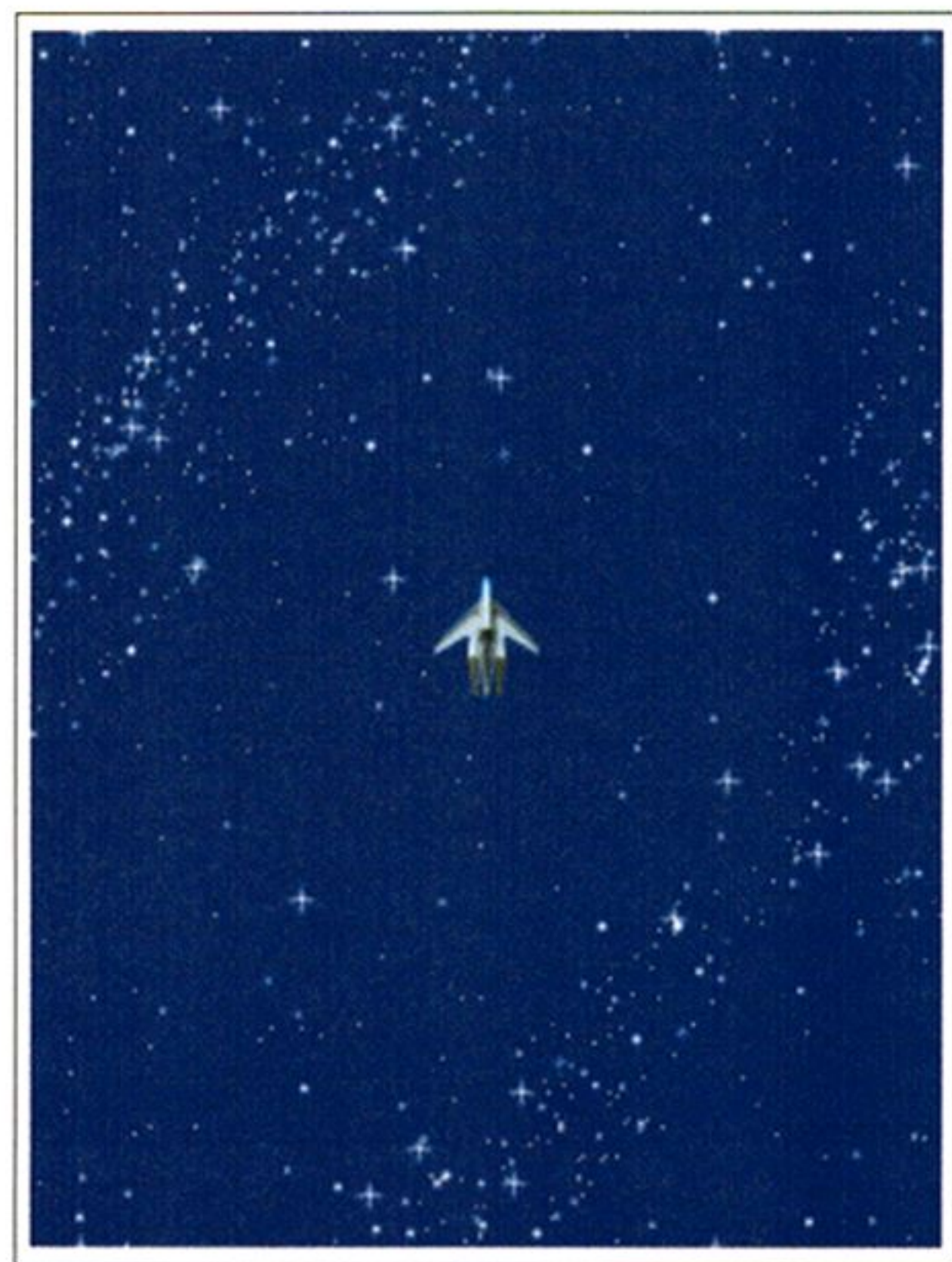



図3 とりあえず自機を動かしてみる

このHTMLファイルをブラウザで開くと，図3のように，星空をバックに戦闘機が表示されるはずだ（サンプルjshooting1）。この戦闘機がスプライト表示によるものだ。

動かそう

当然のごとく，表示はできても止まったままではなんの意味もない。そこで，カーソルキーで上下左右に動かせるようにしてみよう（リスト2）。

キー入力は，KeyListenerインタフェイスを実装することで取ることができる。KeyListenerをimplementsし（5行目），addKeyListenerでリスナを追加する（24行目）。これで，キー入力があった場合に，引数で渡されたthisコンポーネント，つまり自分に対してキーイベントが発行される。さらに，KeyListenerは次の3つのメソッドを持つので，これらも実装する。

```
public void keyPressed ( KeyEvent e )
public void keyReleased ( KeyEvent e )
public void keyTyped ( KeyEvent e )
```

それぞれ，キーが押されたとき，離されたとき，タイプされたときに呼ばれる。キーの種類は引数KeyEventクラスのメソッドgetKeyCode ()で取得でき，たとえばKeyEvent.VK_UPならば，カーソルキーの上キーであることを示す。ここで，boolean型の変数up，down，left，rightに対して，それぞれのキーが押されたときにtrue，離されたらfalseを代入すれば，これらの変数を参照することで，現在キーが押されているかどうかを認識できる。keyTyped ()は今回は使用しないので，空のままでよい。

さて，単にキーどおりに自機を動かすだけなら，キーイベントがあったときに座標を計算し，画面を更新するだけでよいのだが，ここでは先のことを考えて，一定時間ごとに計算・更新を繰り返すようにしておこう。Windowsではタイマを使ったりするが，JavaではもうひとつThreadを立ち上げて，そちらでメインループをぶんぶん回すのが一般的のようだ。

そのためには，まずRunnableインタフェイスをimplementsする（5行目）。そして，start ()，stop ()，run ()メソッドを実装する（70行目以降）。こらららのメソッドの中身は，run ()メソッドのwhile ()ループ内のおのおのの処理を除き，リアルタイムモノのお約束であるので，覚えてしまおう。

そのwhile ()ループの中身だが，まずはThread.sleep (50)で，50ミリ秒スレッドをスリープさせている。つまり，秒間約20回処理が行われる。その下は，キー（の状態を格納した変数）を見て，座標mx，myを計算，111行目でプレーンの移動，再描画という形になっている。難しくはないはずだ。なお，ブラウザで開いたときはアプレットにフォーカスがなく，キー入力を取れないので，いったんアプレットをマウスでクリックしてフォーカスを与えてほしい。

キャラクターをたくさん表示する

ここまでできたら，一気に敵と弾まで表示して，ゲーム性を出してみよう（リスト3）。その前に，今度はちょっとレイアウトしてみることにする。レイアウトというのは，複数のコンポーネントを定義し，それをある規則に従って配置するものだ。Webページでフレームを切るようなものだと思えばよい。あるいは，チャイルドウィンドウと考えてもよい。レイアウトにもいくつかのかたちがあるが，ここではボーダーレイアウトを使うことにする。これは図4のように，最大5個のコンポーネントを配置できるレイアウトだ。すべて使う必要はないので，このうちCenterの部分にゲーム画面を，Northに得点を，Southにはゲームスタートボタンでもつけることにしよう。

このボーダーレイアウトを使うには，まずsetLayout ()でBorderLayoutをセットし（42行目以降），レイアウトしたいコンポーネントをadd ()する。

ただ，これにともない，スプライトマネージャをはじめ，各部を若干修正する必要がある。まず，レイアウトする場合は，SpriteControlではなく，SpriteCanvasを使うこと。クラスメソッド自体はほぼ同じものが実装されている。それにともない，update ()とpaint ()メソッドをオーバーライドする必要がなくなる（SpriteCanvasが内部でやっている）。また，ループの最後でこれまではrepaint ()で再描画させていたのを，SpriteCanvas::Draw ()を呼ぶようにすること。なお，SpriteCanvas::LoadImage ()を呼

んでおくと，いままでやっていたように，イメージがロードされるまでは"Loading..."の文字が表示される。

自分の弾は画面内に5発まで，敵は10機，敵の弾は20発にしておいた。また，今回は背景画像を単に背景に敷くのではなく，バックグラウンドプレーンにしてスクロールさせるので，これもスプライトパターンとして登録する必要がある。あと爆発パターンも含めて，スプライトパターンは6，スプライトプレーンは36（バックグラウンドプレーンは別）となる（41行目）。なお，これらの値は9行目からの文字定数として定義してある。

バックグラウンドのほうは，別途CreateBackground ()メソッドで初期化が必要となる。引数は，確保するバックグラウンドプレーンの数，プレーンの幅，高さ，それにプレーンに貼り付けられるスプライトパターンの数だ。第2引数以降は，それぞれのプレーンに対応する値を配列で指定する。つまり，プレーンによって，サイズや表示できるパターンの数を最適化できる。ここでは大きな背景を1枚貼り付けるだけなので，プレーン数は1，パターン数も1である。

次いで，SetBGPattern ()でスプライトパターンをバックグラウンドプレーンにセットする。引数は，バックグラウンドプレーン番号，バックグラウンドパターン番号，X座標，Y座標，スプライトパターン番号の5つだ。バックグラウンドパターン番号とスプライトパターン番号は混乱しやすいが，前者はCreateBackground ()で設定したパターン数－1を最大値とする，それぞれのプレーン内で管理されるパターン番号で，後者は別途Define ()で定義されたスプライトのパターン番号である。

72行目のaddKeyListener ()だが，今度は頭に"StartBtn."がついている。これはNorthに配置したスタートボタンで発生したキーイベントを，このクラスが受けることを示している。Javaでは，子コンポーネントでイベントを処理しなかったからといって，その親へイベントを送ることはしない。したがって，もし子コンポーネントのイベントをハンドルしたければ，このように明示する必要がある。この場合は，ゲームの開始時には必ずボタンを押すわけで，よそをクリックしたりしなければスタートボタンにフォーカ

North
West
Center
East
South

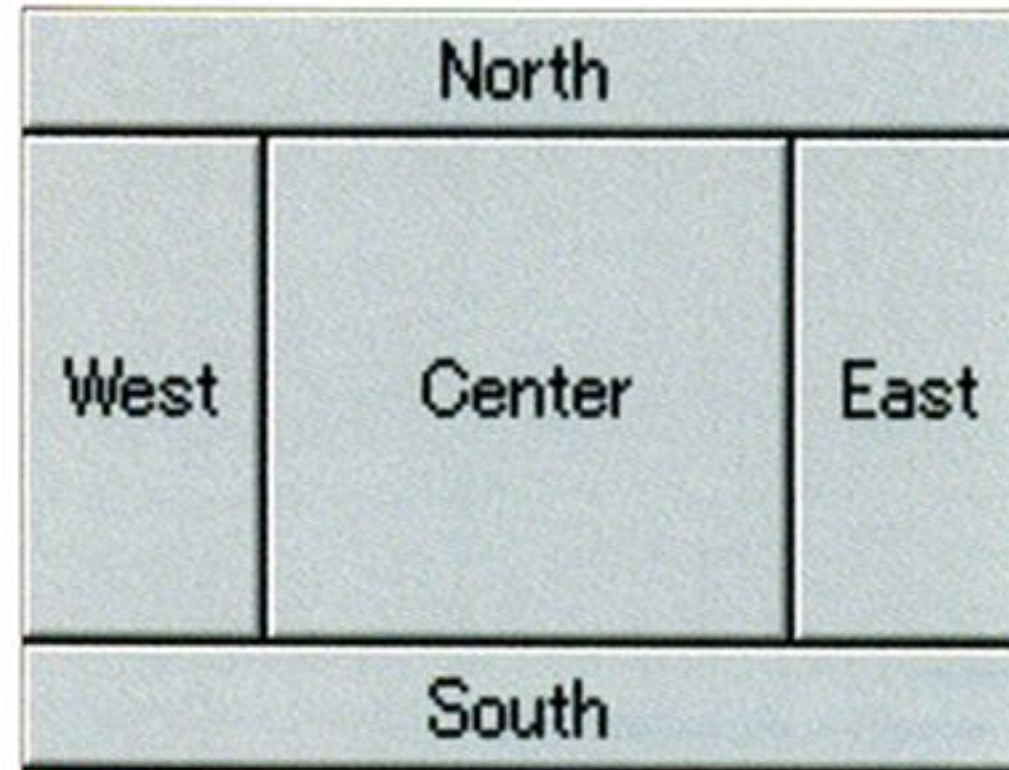



図4　BorderLayout

リスト2

```
[1]import java.applet.Applet;
[2]import java.awt.*;
[3]import java.awt.event.*;
[4]
[5]public class jshooting extends Applet implements KeyListener, Runnable {
[6] static final int WIDTH = 240, HEIGHT = 320; // 背景サイズ
[7] SpriteControl sc; // スプライトコントロール
[8] MediaTracker mt;
[9] boolean up=false, down=false, left=false, right=false;
[10]  int mx=(WIDTH-32)/2, my=(HEIGHT-32)/2;
[11]  Thread thread=null;
[12]  public void init(){
[13]    sc = new SpriteControl( 1, 1, WIDTH, HEIGHT, this );
[14]    mt = new MediaTracker( this );
[15]    Image image = getImage( getDocumentBase(), "img/back.gif" );  // 背景
[16]    mt.addImage( image, 0 );
[17]    sc.SetBGImage( image );
[18]    image = getImage( getDocumentBase(), "img/myship.gif" );  // 自機
[19]    mt.addImage( image, 0 );
[20]    sc.Define( 0, image );
[21]    sc.Set( 0, 0 );
[22]    sc.Move( 0, mx, my );
[23]    sc.Show();
[24]    addKeyListener( this );
[25]  }
[26]  public void update( Graphics g ){
[27]    paint( g );
[28]  }
[29]  public void paint( Graphics g ){
[30]    if( mt.checkID( 0 ) ){
[31]      sc.Display( g, this );
[32]    } else {
[33]      g.drawString( "Loading...", 0, 12 );
[34]    }
[35]  }
[36]  public void keyPressed( KeyEvent e ){
[37]    switch( e.getKeyCode() ){
[38]    case KeyEvent.VK_UP:
[39]      up = true;
[40]      break;
[41]    case KeyEvent.VK_DOWN:
[42]      down = true;
[43]      break;
[44]    case KeyEvent.VK_LEFT:
[45]      left = true;
[46]      break;
[47]    case KeyEvent.VK_RIGHT:
[48]      right = true;
[49]      break;
[50]    }
[51]  }
[52]  public void keyReleased( KeyEvent e ){
[53]    switch( e.getKeyCode() ){
[54]    case KeyEvent.VK_UP:
[55]      up = false;
[56]      break;
[57]    case KeyEvent.VK_DOWN:
[58]      down = false;
[59]      break;
[60]    case KeyEvent.VK_LEFT:
[61]      left = false;
[62]      break;
[63]    case KeyEvent.VK_RIGHT:
[64]      right = false;
[65]      break;
[66]    }
[67]  }
[68]  public void keyTyped( KeyEvent e ){
[69]  }
[70]  public void start(){
[71]    if( thread==null ){
[72]      thread = new Thread( this );
[73]      thread.start();
[74]    }
[75]  }
[76]  public void stop(){
[77]    if( thread!=null ){
[78]      thread = null;
[79]    }
[80]  }
[81]  public void run(){
[82]    try {
[83]      mt.waitForID( 0 );
[84]    } catch( InterruptedException e ){
[85]      return;
[86]    }
[87]    while( thread!=null ){
[88]      try {
[89]        Thread.sleep( 50 );
[90]      } catch( InterruptedException e ){
[91]        break;
[92]      }
[93]      if( up ){
[94]        my -= 4;
[95]        if( my<0 ) my = 0;
[96]      }
[97]      if( down ){
[98]        my += 4;
[99]        if( my>HEIGHT-32 ) my = HEIGHT-32;
[100]     }
[101]     if( left ){
[102]       mx -= 4;
[103]       if( mx<0 ) mx = 0;
[104]     }
[105]     if( right ){
[106]       mx += 4;
[107]       if( mx>WIDTH-32 ) mx = WIDTH-32;
[108]     }
[109]     sc.Move( 0, mx, my );
[110]     repaint();
[111]   }
[112] }
[113]}
```

リスト3

```
[1]import java.applet.Applet;
[2]import java.awt.*;
[3]import java.awt.event.*;
[4]import java.lang.Math;
[5]import java.util.Random;
[6]
[7]public class jshooting extends Applet implements KeyListener, ActionListener, Runnable {
[8] static final int WIDTH = 240, HEIGHT = 320; // 背景サイズ
[9] static final int SHOTNUM = 5; // 弾の数
[10]  static final int ENEMYNUM = 10; // 敵の数
[11]  static final int BULLETNUM = 20;  // 敵弾の数
[12]  static final int SHOT = 1;  // 弾のプレーン番号
[13]  static final int ENEMY = SHOT+SHOTNUM;  // 敵のプレーン番号
[14]  static final int BULLET = ENEMY+ENEMYNUM; // 敵弾のプレーン番号
[15]  SpriteCanvas sc;  // スプライトキャンバス
[16]  MediaTracker mt;
[17]  boolean up=false, down=false, left=false, right=false, zkey=false;
[18]  int mx, my;
[19]  int zkeyblank;
[20]  Thread thread=null;
[21]  Label Score;
[22]  Button StartBtn;
[23]  int score=0;
[24]  int bgwidth[] = { WIDTH };
[25]  int bgheight[] = { HEIGHT };
[26]  int bgnum[] = { 1 };
[27]  int scroll = 0;
[28]  boolean shot[] = new boolean[SHOTNUM];
[29]  int shot_x[] = new int[SHOTNUM], shot_y[] = new int[SHOTNUM];
[30]  boolean enemy[] = new boolean[ENEMYNUM];
[31]  int enemy_x[] = new int[ENEMYNUM], enemy_y[] = new int[ENEMYNUM];
[32]  boolean bullet[] = new boolean[BULLETNUM];
[33]  int bullet_x[] = new int[BULLETNUM], bullet_y[] = new int[BULLETNUM];
[34]  int bullet_dx[] = new int[BULLETNUM], bullet_dy[] = new int[BULLETNUM];
[35]  boolean play = false;
[36]  Random aRandom = new Random();
[37]  Math aMath;
[38]  public void init(){
[39]    // パターン 背景1、自機1、弾1、敵1、敵弾1、爆発1、合計6
[40]    // プレーン 自機1、弾5、敵10、敵弾20、合計36
[41]    sc = new SpriteCanvas( 6, 36, WIDTH, HEIGHT, this );
[42]    setLayout( new BorderLayout() );
[43]    Score = new Label( "SCORE "+score, Label.CENTER );
[44]    StartBtn = new Button( "Start" );
[45]    add( "Center", sc );
[46]    add( "North", Score );
[47]    add( "South", StartBtn );
[48]    mt = new MediaTracker( this );
[49]    Image image = getImage( getDocumentBase(), "img/back.gif" );  // 背景
[50]    mt.addImage( image, 0 );
[51]    sc.Define( 0, image );
[52]    // バックグラウンド1プレーン
[53]    sc.CreateBackground( 1, bgwidth, bgheight, bgnum );
[54]    sc.SetBGPattern( 0, 0, 0, 0, 0 );
[55]    sc.BGShow();
[56]    image = getImage( getDocumentBase(), "img/myship.gif" );  // 自機
[57]    mt.addImage( image, 0 );
[58]    sc.Define( 1, image );
[59]    image = getImage( getDocumentBase(), "img/shot.gif" );  // 弾
[60]    mt.addImage( image, 0 );
[61]    sc.Define( 2, image );
[62]    image = getImage( getDocumentBase(), "img/enemy.gif" ); // 敵
[63]    mt.addImage( image, 0 );
[64]    sc.Define( 3, image );
[65]    image = getImage( getDocumentBase(), "img/bullet.gif" );  // 敵弾
[66]    mt.addImage( image, 0 );
[67]    sc.Define( 4, image );
[68]    image = getImage( getDocumentBase(), "img/bomb.gif" );  // 爆発
[69]    mt.addImage( image, 0 );
[70]    sc.Define( 5, image );
[71]    sc.WaitLoadImage( mt, 0 );
[72]    StartBtn.addKeyListener( this );
[73]    StartBtn.addActionListener( this );
[74]  }
[75]  public void GameStart(){
[76]    int i;
[77]    // 自機のセット
[78]    sc.Set( 0, 1 );
[79]    mx = (WIDTH-32)/2;
[80]    my = HEIGHT-32*2;
[81]    sc.Move( 0, mx, my );
[82]    play = true;
[83]    score = 0;
[84]    sc.Hide();
[85]    for( i=0; i<5; i++ ) shot[i] = false;
[86]    for( i=0; i<10; i++ ) enemy[i] = false;
[87]    for( i=0; i<20; i++ ) bullet[i] = false;
[88]    Score.setText( "SCORE "+score );
[89]    sc.Show( 0, 0 );
[90]  }
[91]  public void keyPressed( KeyEvent e ){
[92]    switch( e.getKeyCode() ){
[93]    case KeyEvent.VK_UP:
[94]      up = true;
[95]      break;
[96]    case KeyEvent.VK_DOWN:
[97]      down = true;
[98]      break;
[99]    case KeyEvent.VK_LEFT:
[100]     left = true;
[101]     break;
[102]   case KeyEvent.VK_RIGHT:
[103]     right = true;
[104]     break;
[105]   case KeyEvent.VK_Z:
[106]     if( zkey==false ){
[107]       zkey = true;
[108]       zkeyblank = 0;
[109]     }
[110]     break;
[111]   }
[112] }
[113] public void keyReleased( KeyEvent e ){
[114]   switch( e.getKeyCode() ){
[115]   case KeyEvent.VK_UP:
[116]     up = false;
[117]     break;
[118]   case KeyEvent.VK_DOWN:
[119]     down = false;
[120]     break;
[121]   case KeyEvent.VK_LEFT:
[122]     left = false;
[123]     break;
[124]   case KeyEvent.VK_RIGHT:
[125]     right = false;
[126]     break;
[127]   case KeyEvent.VK_Z:
[128]     zkey = false;
[129]     break;
[130]   }
[131] }
[132] public void keyTyped( KeyEvent e ){
[133] }
[134] public void actionPerformed( ActionEvent e ){
[135]   if( e.getActionCommand().equalsIgnoreCase( "Start" ) ){
[136]     if( play==false ){  // ゲームスタート
[137]       GameStart();
[138]     }
[139]   }
[140] }
[141] public void start(){
[142]   if( thread==null ){
[143]     thread = new Thread( this );
[144]     thread.start();
[145]   }
[146] }
[147] public void stop(){
[148]   if( thread!=null ){
[149]     thread = null;
[150]   }
[151] }
[152] public void run(){
[153]   int i, j;
[154]   try {
[155]     mt.waitForID( 0 );
[156]   } catch( InterruptedException e ){
[157]     return;
[158]   }
[159]   while( thread!=null ){
[160]     try {
[161]       Thread.sleep( 50 );
[162]     } catch( InterruptedException e ){
[163]       break;
[164]     }
[165]     if( play ){
[166]       if( up ){
[167]         my -= 4;
[168]         if( my<0 ) my = 0;
[169]       }
[170]       if( down ){
```

```
[171]         my += 4;
[172]         if( my>HEIGHT-32 ) my = HEIGHT-32;
[173]       }
[174]       if( left ){
[175]         mx -= 4;
[176]         if( mx<0 ) mx = 0;
[177]       }
[178]       if( right ){
[179]         mx += 4;
[180]         if( mx>WIDTH-32 ) mx = WIDTH-32;
[181]       }
[182]       sc.Move( 0, mx, my );
[183]       // 弾の発射
[184]       if( zkey ){
[185]         // 押しっぱなしのブランク期間
[186]         if( zkeyblank==0 ){
[187]           // 空きを検索
[188]           for( i=0; i<SHOTNUM; i++ ){
[189]             if( shot[i]==false ) break;
[190]           }
[191]           // 弾の設定
[192]           if( i<SHOTNUM ){
[193]             shot_x[i] = mx;
[194]             shot_y[i] = my;
[195]             sc.Set( i+SHOT, 2 );
[196]             sc.Show( i+SHOT, i+SHOT );
[197]             shot[i] = true;
[198]           }
[199]         }
[200]         zkeyblank = (zkeyblank+1)%4;
[201]       }
[202]       // 敵の出現
[203]       if( (aRandom.nextInt()&0xf)==0 ){
[204]         // 空きを検索
[205]         for( i=0; i<ENEMYNUM; i++ ){
[206]           if( enemy[i]==false ) break;
[207]         }
[208]         // 敵の設定
[209]         if( i<ENEMYNUM ){
[210]           enemy_x[i] = Math.abs(aRandom.nextInt())%(WIDTH-32);
[211]           enemy_y[i] = -32;
[212]           sc.Set( i+ENEMY, 3 );
[213]           sc.Show( i+ENEMY, i+ENEMY );
[214]           enemy[i] = true;
[215]         }
[216]       }
[217]       // 敵の移動
[218]       for( i=0; i<ENEMYNUM; i++ ){
[219]         // 表示されている敵の検索
[220]         if( enemy[i] ){
[221]           // 移動
[222]           enemy_y[i] += 4;
[223]           // 画面外判定
[224]           if( enemy_y[i]>=HEIGHT ){
[225]             sc.Hide( i+ENEMY, i+ENEMY );
[226]             enemy[i] = false;
[227]           } else {
[228]             sc.Move( i+ENEMY, enemy_x[i], enemy_y[i] );
[229]             // 敵弾発射
[230]             if( (aRandom.nextInt()&0xf)==0 ){
[231]               // 空きを検索
[232]               for( j=0; j<BULLETNUM; j++ ){
[233]                 if( bullet[j]==false ) break;
[234]               }
[235]               // 敵弾の設定
[236]               if( j<BULLETNUM ){
[237]                 // 背後からの攻撃は禁止
[238]                 if( my>enemy_y[i] ){
[239]                         double l = aMath.sqrt((mx-enemy_x[i])*(mx-enemy_x[i])+(my-
enemy_y[i])*(my-enemy_y[i]));
[240]                   // 至近距離からの発射は禁止
[241]                   if( l>100.0 ){
[242]                     bullet_x[j] = enemy_x[i];
[243]                     bullet_y[j] = enemy_y[i];
[244]                     bullet_dx[j] = (int)aMath.round( 6*(mx-enemy_x[i])/l );
[245]                     bullet_dy[j] = (int)aMath.round( 6*(my-enemy_y[i])/l );
[246]                     sc.Set( j+BULLET, 4 );
[247]                     sc.Show( j+BULLET, j+BULLET );
[248]                     bullet[j] = true;
[249]                   }
[250]                 }
[251]               }
[252]             }
[253]             // 自機との当り判定
[254]             if( enemy_x[i]>mx-18 && enemy_x[i]<mx+18 ){
[255]               if( enemy_y[i]>my-18 && enemy_y[i]<my+18 ){
[256]                 // 敵爆発
[257]                 sc.Set( i+ENEMY, 5 );
[258]                 // 自機爆発
[259]                 sc.Set( 0, 5 );
[260]                 enemy[i] = false;
[261]                 score += 10;
[262]                 Score.setText( "SCORE "+score );
[263]                 play = false;
[264]               }
[265]             }
[266]           }
[267]         } else {
[268]           sc.Hide( i+ENEMY, i+ENEMY );
[269]         }
[270]       }
[271]       // 弾の移動
[272]       for( i=0; i<SHOTNUM; i++ ){
[273]         // 表示されている弾の検索
[274]         if( shot[i] ){
[275]           // 移動
[276]           shot_y[i] -= 16;
[277]           // 画面外判定
[278]           if( shot_y[i]<=-32 ){
[279]             sc.Hide( i+SHOT, i+SHOT );
[280]             shot[i] = false;
[281]           } else {
[282]             sc.Move( i+SHOT, shot_x[i], shot_y[i] );
[283]             // 敵との当り判定
[284]             for( j=0; j<ENEMYNUM; j++ ){
[285]               if( enemy[j] ){
[286]                 if( shot_x[i]>enemy_x[j]-12 && shot_x[i]<enemy_x[j]+12 ){
[287]                   if( shot_y[i]>enemy_y[j]-12 && shot_y[i]<enemy_y[j]+12 ){
[288]                     // 弾消去
[289]                     sc.Hide( i+SHOT, i+SHOT );
[290]                     shot[i] = false;
[291]                     // 敵爆発
[292]                     sc.Set( j+ENEMY, 5 );
[293]                     enemy[j] = false;
[294]                     score += 10;
[295]                     Score.setText( "SCORE "+score );
[296]                   }
[297]                 }
[298]               }
[299]             }
[300]           }
[301]         }
[302]       }
[303]       // 敵弾の移動
[304]       for( i=0; i<BULLETNUM; i++ ){
[305]         // 表示されている敵弾の検索
[306]         if( bullet[i] ){
[307]           // 移動
[308]           bullet_x[i] += bullet_dx[i];
[309]           bullet_y[i] += bullet_dy[i];
[310]           // 画面外判定
[311]           if( bullet_x[i]<=-32 || bullet_x[i]>=WIDTH ||
[312]             bullet_y[i]<=-32 || bullet_y[i]>=HEIGHT ){
[313]             sc.Hide( i+BULLET, i+BULLET );
[314]             bullet[i] = false;
[315]           } else {
[316]             sc.Move( i+BULLET, bullet_x[i], bullet_y[i] );
[317]             if( bullet_x[i]>mx-12 && bullet_x[i]<mx+12 ){
[318]               if( bullet_y[i]>my-12 && bullet_y[i]<my+12 ){
[319]                 // 敵弾消去
[320]                 sc.Hide( i+BULLET, i+BULLET );
[321]                 // 自機爆発
[322]                 sc.Set( 0, 5 );
[323]                 play = false;
[324]               }
[325]             }
[326]           }
[327]         }
[328]       }
[329]       // 背景スクロール
[330]       scroll += 2;
[331]       if( scroll>=HEIGHT ) scroll = 0;
[332]       sc.BGScroll( 0, 0, scroll );
[333]     }
[334]     sc.Draw();  // スプライト描画
[335]   }
[336] }
[337]}
```

SCORE 270

Start

図5 弾やキャラクターをたくさん出してみる

STAGE 1 SCORE 1360 LEFT 1

Start

図6 名もなかった横シュー

スがあるはずということで，スタートボタンのイベントだけを取っている。もし不安ならば，片っ端からaddKeyListener ()するといいだろう。逆に，73行目のaddActionListener ()は，スタートボタンが押されたアクションをハンドルするものである。少なくともこのプログラムでは，スタートボタンからしかアクションはないので，StartBtnクラスだけでよい。

そのアクションイベントだが，ActionListenerをimplementsし(7行目)，

public void actionPerformed (ActionEvent e)

を実装することで取ることができる(134行目)。ActionEvent::getActionCommand ()は，アクションが起こったコマンドの文字列(Stringクラス)を返す。"Start"というキャプションがついたボタンならば，その名前が格納されたStringクラスのインスタンスとなる。それを，String::equalsIgnoreCase ()で判定している。大文字／小文字を無視した，文字列比較メソッドだ。ここではアクションイベントはスタートボタンを押した場合しかこないはずだが，念のため。なお，playというのはboolean型の変数で，ゲーム中のときはtrue，そうでないときはfalseが格納されている。

さて，スタートボタンが押されると，GameStart ()メソッドが呼ばれて，もろもろの初期化が行われるわけだが，それぞれのキャラクター用に変数を用意しなければならない。たとえば，自機の弾の管理。表示されているかどうか，およびその座標(28行目以降)。配列でそれぞれ個別に確保しているが，こんなときはやっぱり構造体がほしいと思う。クラスにしてしまって，動きや当たり判定のメソッドを実装，さらに種類を増やすときにはそれを派生させれば，プログラム的には美しいかもしれないが，今回はクラスで難しいことはしないという約束なので。敵の動きも今回は真下に飛んでくるだけという安易なものなので，必要な変数は弾と同じ。敵の弾だけは，自分に向かって飛んでこなければゲームにならないので，変移を格納する変数も用意した。

あとの処理は，すべてrun ()メソッドの中だ。見事なスパゲティ(死後)なので，少々階層が深くなってうざったいかもしれないが，ゲームプログラミングとしては当たり前のことしかしていないので，難しくはないだろう(あまりよくない例ではある)。細々と説明しているときりがないが，筆者にしてはコメントを多く入れたほうなので，その辺りを手がかりになにをしているのか理解してほしい。新しく出てきたメソッドだけは紹介しておこう。

int Random::nextInt ()
　乱数値の発生
int Math::abs (int)
　絶対値
double Math::sqrt (int)
　平方根
long Math::round (double)
　近似整数(丸め)

キャラクターの動きなどを演算後，最後にバックグラウンドをスクロールさせている(329行目以降)。scrollという変数で，1フレーム当たり2ドットずつオフセットをずらし，BGScroll ()メソッドで表示位置をずらす。第1引数はプレーン番号，次いでXオフセット，Yオフセットだ。

このようにして完成したのが図5だ(サンプルjshooting3)。なんかスプライトマネージャに同梱のサンプルShootSaucerと大差なくなってしまったが，ここから先はJavaでもスプライトマネージャでもなく，プログラミングのセンスなので，各自でバリバリのカッコイイゲームを作ってもらいたい。本当は，大昔に某誌で作った横スクロールゲームの紹介もしようかと思っていたのだが(図6)，これがとあるところで使われることになってしまったので，おいそれとソースを出すことができなくなってしまった。今回のものに比べると，マップとかパワーアップとかボスとかあって，遥かにゲームらしかったんだが。いま考えると，あんな本(失礼)でなんでそんなに気合入れて作ってたんだろうとも思うが，あの頃は仕事がなくて暇だったんだね，多分。数カ月にわたって，連載で作ったもんだし。どうしてもソースがほしいという人は，昔の雑誌を探してほしい。出典は内緒だ(それじゃわかんねぇって)。

Javaゲー大量生産予定

今回のことは，筆者にも大変勉強になった。なにせ，キーイベントの取り方が変わったことも，Thread.stop ()を使ってはいけなくなったことも知らなかったくらいだから。随分離れていたにもかかわらず，また舞い戻ってきたのは，冒頭でもちょっと述べたように，仕事でまた必要になったからだ。いろいろと(というほどでもないか)経緯があって，for Gamer Net (http://www.4gamer.net/)でJavaゲーを掲載することになったのだ。月に2本ペースで新作を作れ(！)という無茶な要求に対して，ギャラは普通では考えられないくらい安いのだが，「ちょっと楽しいかもね」ってことで受けちゃったのが運の尽き。ま，暇があったら覗いてもらって，Javaでぽちぽちと遊んでもらって，「そういやスプライトマネージャってので作られているんだっけ」と思い出してもらえれば幸いである。

特集

ポケットの中のコンピュータたち

～携帯系プログラミング～

携帯機器系のプログラミング特集をやるというのは２号くらい前（１年半から２年くらい前か……）からあてになりそうにない予告に挙がっていたわけだが，21世紀を迎えてようやく実現することになった。世の中の流れを思い返すと，企画当初に比べていくつかの分野はめざましく進展し，いくつかの分野は思ったよりは進歩しなかったようだ。１年前前にやっておかなければいけなかった特集なのだが，新しいトピックはさほど多くない。

本命はiモードなどの携帯電話系機器とされているものの，まだまだなにが出てくるかわからない。そういった状況も特に変わってはいない。

携帯機器というと，機能が低そうでメモリ容量などせこせこしたプログラミングしかできないのではないかと思い込んでいる人も多いかもしれない。確かにPCに比べると速度も遅く，メモリ容量も少ない。だが，PalmのCPUやメモリ容量を聞くと驚く人も多い。基本的にCPUやメモリ容量ではX68000XVIかX68030クラスのキャパシティはあるるのだ（メモリは，しこたま遅いのだが）。その気になれば，相当のものは動くはずである。

いつでも身に着けていられる環境であるからこそ，自分の思うままにしていきたい。そういうものではないだろうか。

携帯デバイスに広がるプログラミング環境

中野 修一　Nakano Shuichi

PalmやVisorの積極的展開で活況を呈するPDA機器。はたまた「ケータイ」という単語で通用する携帯電話系デバイスは現世代から次世代の最重要メディアだ。それら「小さな箱庭」の動向と特徴を押さえておこう。

携帯機器の特徴はといえば，なにより「小さい」ということだ。なりは小さくても，WindowsCE関係とかでは搭載しているCPUなどは結構凄かったりもするのだが，主に表示能力と入力デバイス，外部記憶という面ではかなり制約を受ける。電池駆動なので，省電力設計などでCPUクロックが一定でない可能性があるなど，プログラムを組む側からすると気をつけなければならない点もいつくかあるだろう。

なにに使われているのか？

まず，こういった携帯機器がなにに使われているのかと調べてみると，Palm系のユーザーは「住所録！」と力強く答えてくれた。いわゆる「PDA」と呼ばれている類のものでは多かれ少なかれ似たようなものだろう。

携帯電話系では，もちろん電話としての機能とメールが主流となる。キーボードつきのWindowsCE機などではパソコンの代替品としてのあらゆる用途が想定されているといっていいだろうが，やはり住所録やメールチェックなどが主流だろうし，携帯ゲーム機関係では，当然ながらゲームということになる。ゲーム機では，それ以外のことはほとんどできないのも特徴といえるだろう。

電話やゲームといったもの以外では，なにがどれくらいの比重で重視されるのだろうか。

たとえば，インターネット接続などが携帯端末からできなければならないとするのなら，当然，携帯電話系の機能はそれぞれの機種が備えておかなくてはならない。あるいは，Pinコンパクトのようなモバイル通信機能が簡単に使えるようになっている必要がある。

ざっと見た感じでは通信機能は必須とはされていないような感じだ。インターネットへどこからでもアクセスできる環境というのは非常に便利なものだろうが，Webへちゃんとアクセスしようとすると，それなりのハード/ソフト構成が必要になる。メールがチェックできれば十分とするなら，携帯電話でこと足りることにもなってしまう。実際，それで十分なような気はするが。

結局のところ，パソコンの補佐ツールとしてデータを持ち出したり，登録してあるデータをブラウズするという電子メモ的な用途がいちばん重要になるようだ。あとは，ちょこちょことしたツールが動いてくれればそれだけでも便利に使える。常時携帯できるコンピュータ環境というのはそれなりに魅力的なのだ。

さまざまな携帯機器

さて，ひと口に携帯機器といっても，携帯電話からキーボードつき端末までいろいろあって，ひと筋縄ではいかない。いくつかの切り口で切り分けていくことになるのだが，機能という点で見るともっとも特徴的に携帯機器がいくつかのパターンに分かれることがわかると思う。ザウルスとPalmを考えるともっともわかりやすい。

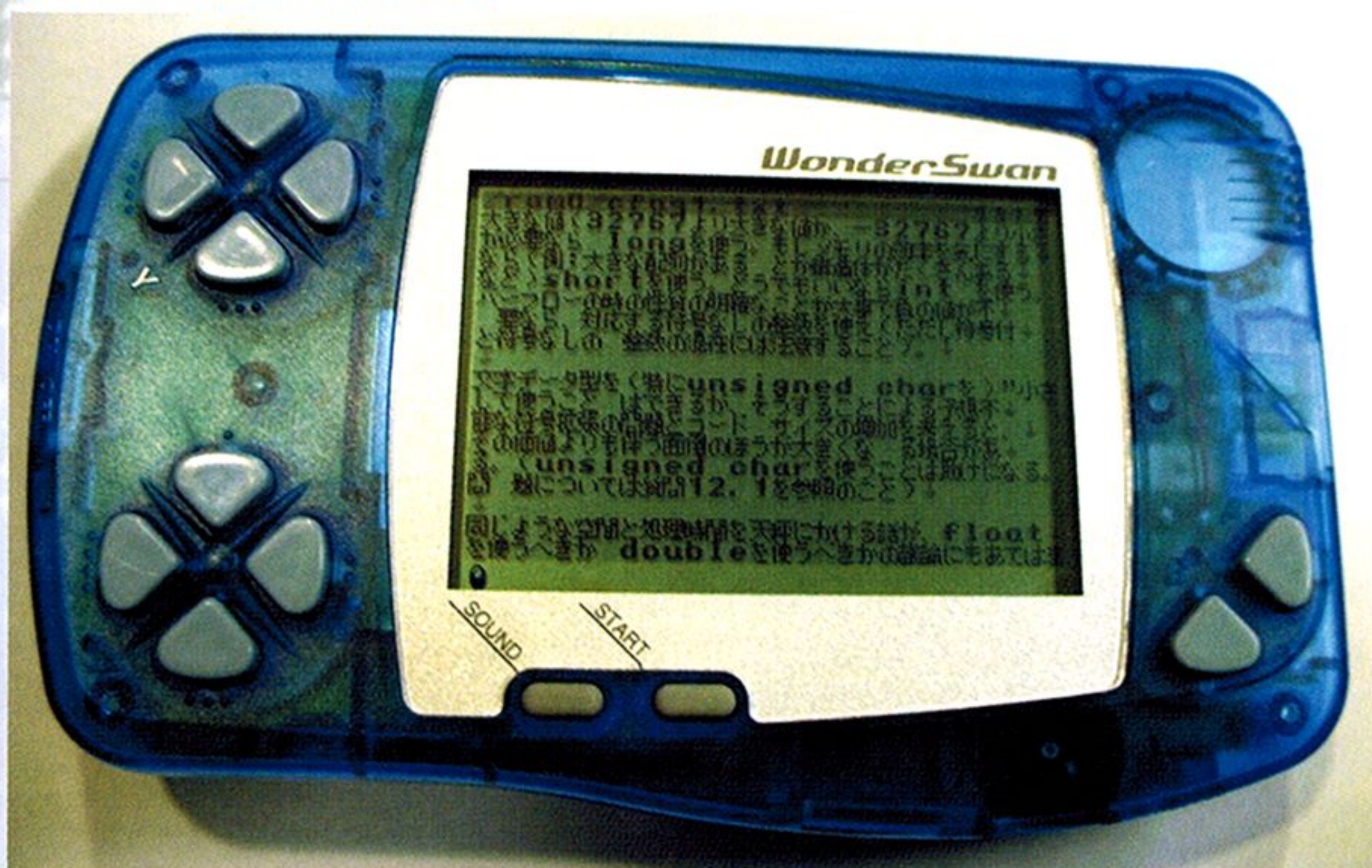
ザウルスは単体でさまざまなことができる（まあ，ビデオ関係では周辺機器が必要だが）。機能的にはほとんどのものが単体で完結している。それゆえに，たとえばデータ入力などで使いにくいと本質名な欠陥ともなろう。幸い，ザウルスの手書き文字による日本語認識は業界トップクラスであり，PDAとしての完成度も高い。

対するPalmはPalmwareの装備でいろいろできなくはないのだが，基本的にはパソコンで入力したデータを出先で閲覧するということが主となるような構成になっている。データ処理もできなくはないのだが，さほど得意ではないし，データ入力ができなくもないが，かなり苦手なほうといえるだろう。

なんでもできるようにするにはそれなりのCPUパワーやメモリを要求されることになる。ザウルスはPalmより遙かに高性能なマシンであり，ついでにいうとWindowsCE機の場合はもうちょっとパソコンとの連携が強いが，基本的には単体でほぼなんでもできる。

極論するとパソコンとほぼ同等の機能が必要だが，そうなると価格や大きさなどでかなり困難な選択を強いられることになる。これは得策ではない。もっとも用途の集中するデータブラウズとメールチェックなどの機能に抑えたほうが携帯機器としてはリーズナブルではないかと考える人がいるのも当然だろう。実際Palmはさほど強力なハードウェアではないが，そういう考え方でもっとも成功を収めている。さらにいえば，CPUにはもっとも強力なものを使用しているWindowsCE系のマシンは決して成功しているとはいえない。

Palmはグラフィティという入力方式に慣れることができるかいなかで評価は大きく分かれるのだが，かなりグラフィティを使いこ

図1　WonderSwanでのテキスト表示例

なしている人でさえなお，データ入力はパソコン上で行うのが当たり前という認識をしていることが多いようだ。ほかに方法がないとなれば，多少我慢してそれを覚えようともするのだが（携帯電話でせっせとメールを書いている女子高生は忍耐強いと思う），多くの人にとって，出先でのデータ入力の必要性というのはさほど多くない。大半はデータの確認だ。

ブラウズに特化という方策は非常に有効に思える。

であれば，WonderSwanでもいんじゃないか？　と考えるのは間違いだろうか？　恵理沙フォントでのテキスト表示サンプルを見るとたいていの人は「おぉ！」と唸る。表示メディアとしての潜在能力はなかなかのものだ。WonderWitchのテキスト表示サンプルでは基本的に2階調しか使っていない。WonderSwanの液晶は階調性が悪いのであまり理論値どおりにはいかないのだが，適切にガンマを設定したうえで4階調フォントを導入すれば非常に魅力的なテキスト表示環境ができてくると期待される。

しかしWonderSwanは，現状ではデータ入力環境としてははなはだ貧弱だ。入力のためのインタフェイスがゲーム用のものしか用意されていないので，データはもっぱらパソコンからの転送に限定するか，それでなければテキスト入力などは入力プログラムから作らないといけない。それは楽ではない。せめてボタンがもういくつかあれば全然違ったものになるのにと残念だ。ほかのPDA並にタッチパネルなどがあれば新しい道も開けてくるだろうに。

バンダイはWonderSwanColorの発表の際に，「もうひとつのケイタイ」というコンセプトを打ち出した。ある意味，PDA路線も目指すという意思表示のようにも思われるのだが，実質が伴っていない。販売戦略も失敗しているようで，実際あまり売る気がないのが残念だ（秋葉原でも売ってないってのはちょっと問題ありすぎだろう）。入力に関しては当座はあきらめて簡易入力ですませ，最悪，漢字コード表からの選択でもかまわないんじゃないかと思っている（オムロンはWonderGate用にはモバイルWnnを作っているのだけれど）。

Oh!Xとしては，PDA的なツールも作りたいのだが，もちろんゲームも忘れてはいけない。となると，データブラウズに特化されたようなシステムよりは，ザウルスやWindowsCE系のパワフルなデバイスのほうが向いているといえるかもしれない。もちろんザウルスもWindowsCE系マシンも開発環境は提供されている。今回ザウルス関係の開発記事があがっていないのが残念なのだが，基本的にはCodeWarriorで開発することになるだろう（世の中，CodeWarriorで開発できない機種のほうが少ないのだろうけど）。CodeWarrior以外にシャープ純正の開発環境も存在はするが，現在ではCodeWarriorをメインに押し出していることなどから，もはやCodeWarrior中心に考えておいたほうがいいだろう。

シャープ純正開発環境SZAB試用版のダウンロードは以下のURLで可能だ。

http://more.sbc.co.jp/downlord/szabtrial421-200104/szabtrial.asp

ゲーム機はゲーム専用のハードウェアを搭載しているので，汎用PDAなどよりは簡単にゲームが作れる。多少スペックが低くてもまったく問題はない。やはり軽いスプライトは便利だ。WonderSwanは開かれたプログラミング環境を提供している現状で唯一のゲーム専用機である。Pdaとして見るとゲームが作りやすい唯一の環境ともいえる。

携帯電話系という選択肢

昨年末にようやくiモード用Javaの仕様が発表された。待望の携帯プログラミング環境である。

http://www.nttdocomo.co.jp/mc-user/i/java/

で詳細な仕様が手に入るので，興味のある方はぜひドキュメントを入手しておいてほしい。一般の人がほとんどコストをかけずに開発でき，ユーザー数も極大という意味では最強の携帯プログラミング環境となる。

どのようなAPIがサポートされているかというのはマニュアルを読んでいただくとして，懸念されたメモリ容量だが，プログラム容量で10Kバイト，データ領域で5Kバイトということに正式決定した。JARに固めた状態で10KBということなので，あまり大きなプログラムは実行できないにしてもどういう規模のものなら大丈夫かというのを見極めることは重要だ。ちなみに，JAR（JAVA ARCHIVE）の圧縮形式はZIPと同等といわれている。ということで，バイトコードでもうまくすれば半分近くまで圧縮できるかもしれない（もちろん過度な期待は禁物だ）。

具体的に解説すると，電話機に入っているメモリが15Kバイトだということではない。各端末はそれぞれアプリケーション用のメモリを持っていて，ひとつのJavaアプレットが使用できるプログラムサイズが10Kバイト，そのプログラムに割り当てられるスクラッチパッド（ローカル環境の永続的なメモリ空間。ストレージデバイスと考えておけばよいだろう）が5Kバイトに制限されるということだ。メーカーによってはより大きなスラクラッチパッドサイズをサポートすることもあるようだが，互換性上は問題がある。

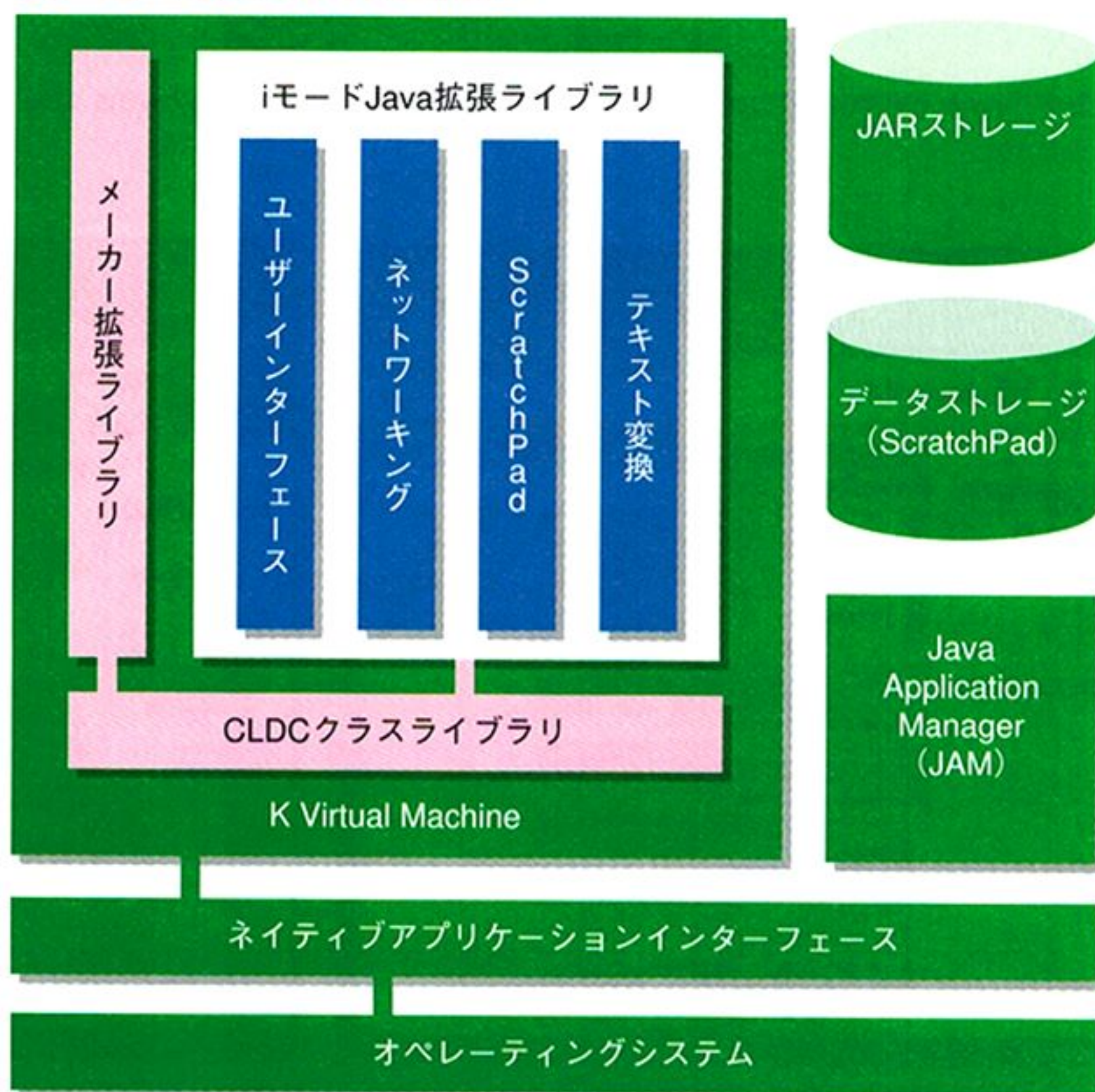
アプリケーションメモリ内にはJavaプログラムを複数持つことができるのだが，ひとつのプログラムを大きくということはできない。プログラムサイズについては，10Kバイトを超えると正常なダウンロードが保証されなくなる。一括してプログラムをダウンロードされてローカル環境だけで使われてしまうと電話会社は商売上がったりになるだろうし，ゲームオーバーごととか，面クリアごとにサーバにアクセスして新しいプログラムをダウンロードしてくるような仕組みになるのはしかたないのかもしれない。まあ，10Kバイト単位という仕様だと割り切るしかない。

なお，セキュリティなどの絡みもあって同時に動くジョブはひとつに限定されているので，複数のプログラムをメモリ上に置いておいてもダウンロード時間の節約以上のメリットはない（マルチタスクできないという意味ではないので注意）。

メモリの節約が携帯Javaでの最大の課題だ。どのようにしてプログラム以外のデータ部分を除くか（JARに含めず起動後にスクラッチパッドにダウンロード）など，いくつかの指針がJavaコンテンツ開発ガイドに記載されているので一読しておこう。これは容量や画面に制限のある携帯系機器プログラミング全般にとって参考になる部分もある。

W-CDMAによるブロードバンド化を春に控え，Java端末自体も春モデルからが本命といわれている。あまりあせって導入する必要はないのだろうが，春以降は各種サービスが本格的に稼働して早い者勝ち状態になるのだろう。

先日中森章氏とCPUの動向について話をしてたら，最近の流行はJava拡張だそうで，世の中まだまだそちらの方向に突進する気配が濃厚だ。今後しばらくはJavaをやっといたほうがよいのだろう。10Kバイトと非常に狭めの箱庭だが，それだけに手の掛け甲斐があるというものかもしれない。

図2　iモードJavaのシステム構成図

eMbedded Visual Tools 3.0でWindowsCEプログラミング

大和哲　Yamato Satoshi

いつも使っているPCの環境にいちばん近いPDA，それがWindowsCE環境です。登場時はコストパフォーマンスにかなり問題があったのですが，最近では比較的こなれた値段で本体を入手することも可能です（ほかのものと比べるとまだ高めですが）。プログラミングの手軽さは際立っており，再評価してもよいシステムなのかもしれません。

携帯機器プログラミングといえば，やはりPDAである。PDAといえば，Palm，Visor，CLIEといったPalm系，それにシャープのZaurousの人気が強いが，プログラミングができるということであればWindowsCE機の存在を忘れてはいけない。カシオのCasiopea E-700，NECのモバイルギアIIシリーズ，CompaqのAeroシリーズなどいろいろな機種が発売されている。

以前は性能に比べて，あるいはPalm系などのマシンと比較してもリーズナブルな価格の機械のなかったCE機だが，最近はNTTドコモのから発売されている「シグマリオン」や「G-FORT」などが，東京近辺だと5万円も出せばそれなりのものが手に入るようになったのだ。

それに，本体価格だけでなく，WindowsCEに関しては開発環境もかなりよい状況になってきた。もともとWindowsCEは，おそらく現在の多くのプログラマの共通プラットフォームであるWindowsプログラマにとって，とてもプログラミングがしやすい環境となっているのだが，最近ではプログラム環境をとても簡単に安価に手に入れることができるようになったのだ。

Windowsでは，アプリケーション開発に必要なコンパイラなどが含まれた「Visual Studio」というソフトが販売されているが，WindowsCEには「eMbedded Visual Tools」というソフトが販売されている。実は，これまではWindowsCE用のソフトを開発するためには汎用のCコンパイラとWindowsCE用のSDKが必要であったため，WIN32用のVisualStudioを購入し，さらに WindowsCE Toolkit を入手するという出費と手間を強いられていたわけだが，これが「WindowsCE eMbedded Visual Tools」というWindowsCE専用のツール単独で開発できるようになった。NT系のOSを使っているのなら，WindowsCEのエミュレータも含まれているので，実機を使わずにプログラミングをすることすら，このソフトだけで可能になったのだ。

さらに都合がいいのは，この「eMbedded Visual Tools 3.0」はとても安価に入手することができる，ということだ。

http://www.microsoft.com/mobile/developer/BeginnerDev/tools.asp

このソフトはこのURL（米国マイクロソフトのWebページ）から購入することができ，価格自体はゼロ，日本への発送の場合，送料として14.95ドルだけ支払えば手に入れることができるのだ。

さらにうれしいことに，購入ページを見るとわかるが，実はこのeMbedded Visual Toolsには日本語ローカライズされたバージョンも存在する（上に挙げたのは日本語版の購入ページのURLだ）。この日本語版は，インストーラ，ツールはほぼ完璧に日本語化され，ヘルプも主要な部分はかなりが日本語に翻訳されており，当然，オブジェクトも日本語，英語両方のものが作れるわけだ。

eMbedded Visual Toolsの中身

このeMbedded Visual Tools 3.0は，正確にはPocketPCやH/PC2000などH/PC Pro2.0以降のWindowsCE機のプログラミングをするための統合環境だ。PocketPC，H/PCというのはWindowsCEを搭載した機種の別称で，形状やOSバージョンによって名前がつけられている。大別すると，

H/PC（ハンドヘルドPC）
　キーボードを持つノートパソコンよりひと周り小型のマシン
H/PC WindowsCE 1.0
　H/PC 2.0 WindowsCE 2.0ベース
H/PC Pro, 3.0
　WindowsCE 2.11ベース。VGA/SVGA をサポート，65536色対応。
　Pocket Access搭載
H/PC2000 WindowsCE3.0ベース

P/PC,PocketPC　キーボードを持つたないPDAタイプのマシン
PalmSize PC　Windows CE 2.0ベース
Palm-size PC Version1.2 カラー版 Palm-sizePC
PocketPC WindowsCE3.0ベース

図1　eMbedded Visual Toolsの日本語ページ

図2　ご覧のように価格は＄0.00!

ということで，簡単にいえば，このeMbedded Visual Tools 3.0は WindowsCE 2.11以降のマシン用の開発ツールだということになる。このなかには，WindowsNT4.0（要SP5,IE5.01）/2000/98SE/Me上でクロス開発するためのツールがひととおりそろっている。

内容的には，まずはクロスコンパイラがふたつ。

・eMbedded Visual BASIC
・eMbedded Visual C++

ほぼフルセットのVisual BASICとVisual C++の機能で，オブジェクトをMIPS,SuperH,ARM,それにx86エミュレータ用として作ることができる。SDK

としては，.Microsoft Windows Platform SDK for Pocket PC,

・Windows CE Platform SDK（H/PC Pro 3.0）
・Windows CE Platform SDK（Palm-size PC 1.2）

が含まれている。H/PC SDKに含まれるH/PC 2.0用ライブラリを組み込めば，H/PC 2.0用のにもプログラムを作ることもできるらしい（公式にはサポートされていない）。

　また，これらのデバック環境として，

・H/PC Proエミュレータ
・PocketPCエミュレータ

が添付されている。WindowsNTおよび2000上でこれらのOSのエミュレーションができるのだ。このエミュレータが実にWindowsCEの動作をよくエミュレートしていて，たとえばスタートメニューやディスク構成もWindowsCE機のそれと同じだし，PocketWordやPocketExcelもちゃんと動く。オフラインモードであればInternetExplorerさえ使える。もし，まだWindowsCE機を持っていないが，H/PCなどに興味があるのだったら，迷わず，このツールを入手してCE体験することをおすすめしたいくらいだと，筆者は思う。

　ちなみに，このエミュレータ上で動くオブジェクトはx86上でのエミュレータ用にコンパイルされたものを使う。また，残念ながらWindows98SEではこのエミュレータは利用できない。

　なお，公式にはサポートされていないが，H/PC2.0用のSDKを前のToolKitより別途インストールすると，「eMbedded Visual Tools 3.0」でもH/PC 2.0用のソフトが作れるそうである（ただし，これをやると，H/PC Pro3.0用のSDKやPalm-size PC 1.2用のSDKを再度インストールしなおす必要があるはずだ。実行は自己責任で）。

eMbedded Visual BASICでのプログラミング

　さて，それでは，このeMbedded Visual Toolsに含まれているツールを使ってプログラムを試してみよう。まずはeMbedded Visual BASIC 3.0を使ってのプログラミングだ。

　これはWindowsCE用のオブジェクトを生成するVisual BASICで，WindowsのVisual Basic のデスクトップ バージョンで使われる言語のサブセットとなっている。機能的には，使用できるデータ型はバリアント（Variant）型のみであるなどの制約はあるが，Windows版でも定評のある，「IDEのウィンドウ内でフォームを作っていって，目で見ながら開発できる」プログラミングのしやすさはWindowsCEでもそのままだ。

　VisualBASICのプログラミングでは，まず，新規にフォームを作る。そのうえにボタンやリストボックスなどのコンポーネントを載せていき，それぞれのコンポーネントやフォームに，それぞれのイベントに対応した動作を書いていく。

　たとえば，メインになるフォームの上にボタンをつけて，そのボタンがおされたら，文字入力部分の表示を変えるのであれば，IDE画面上で新規のフォームを作り，ボタンをその上に作る。ボタンをダブルクリックすると，エディタが現れて，

```
Private Sub Command11_Click ()

End Sub
```

のように「ボタン1がクリックされた」イベントをハンドルする部分のプログラムリストが表示される。ここに「表示を変える部分」，たとえば，

```
Private Sub Command11_Click ()
   Text1.Text = Command11.Caption
End Sub
```

のようにそれに該当する動作を書いていけばいいだけである。

というわけで小ネタだが，作ってみたのが，リスト1「割り勘電卓」。フィールドに合計金額を入れて，人数ボタンを押すと，割り勘した場合の一人あたりの金額が出てくるというものだ。はっきりいえば，作り方は，フォーム上に文字入力フィールドとボタンをいくつか貼り付けて，ボタンの人数に応じて，

```
Text1.Text = Int (Text1.Text) / (人数)
```

と書いただけなので，プログラム的にはまったく手間がかかってない。実は筆者は，Visual BASICについてはほとんど触ったことがないのだが，それでも15分程度で作れてしまった（15分のうち，ほとんどはVisual BASICの文の末尾には「;」がいるのかどうか調べていただけである）。

　それでもWindowsプログラミングだと，たとえばCであれば，まずウィンドウの上にボタンを配置して……という部分のプログラムから作らなくてはならないわけだから，それに比べると脅威といえるくらいの簡単さだ。

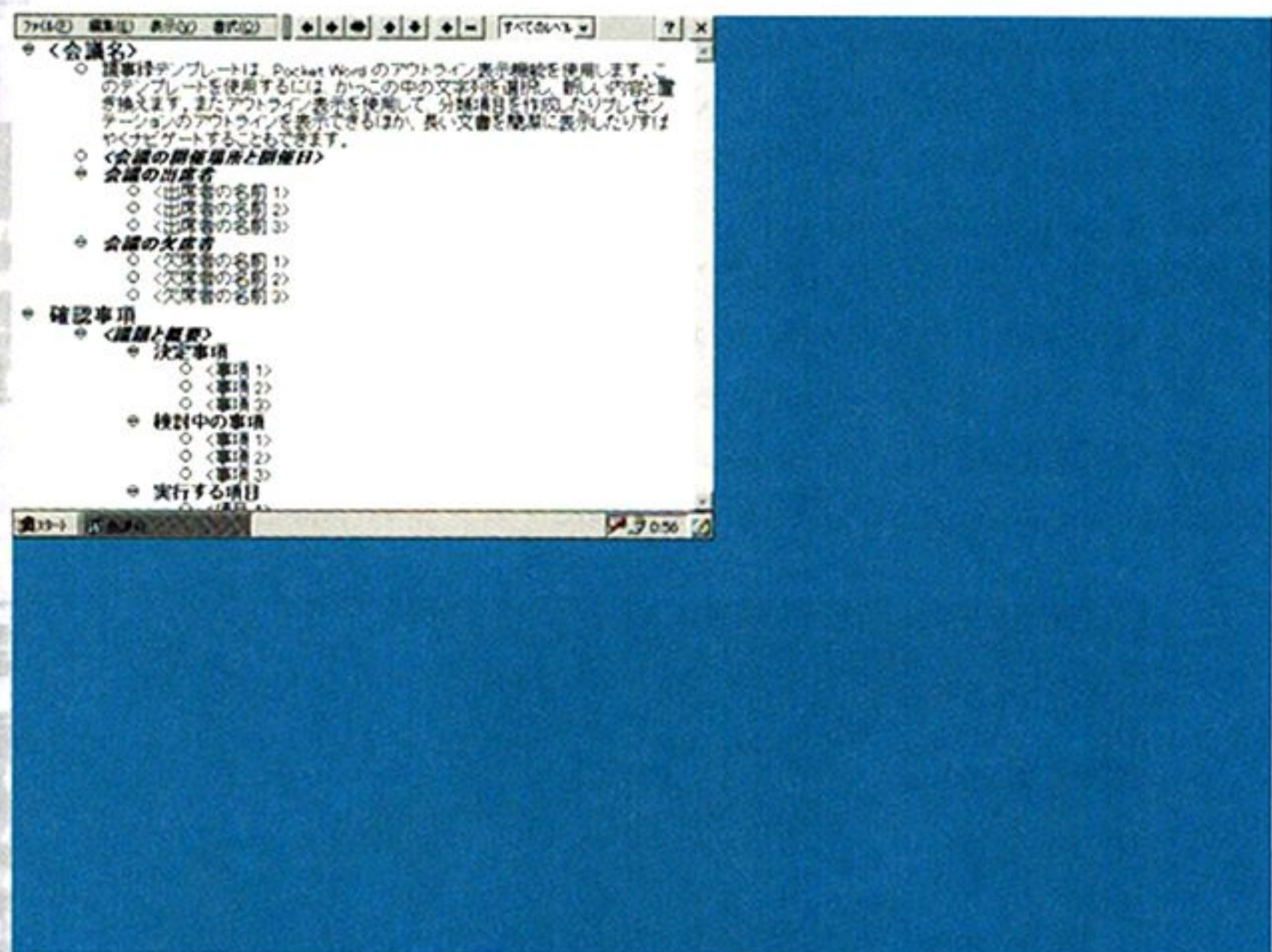
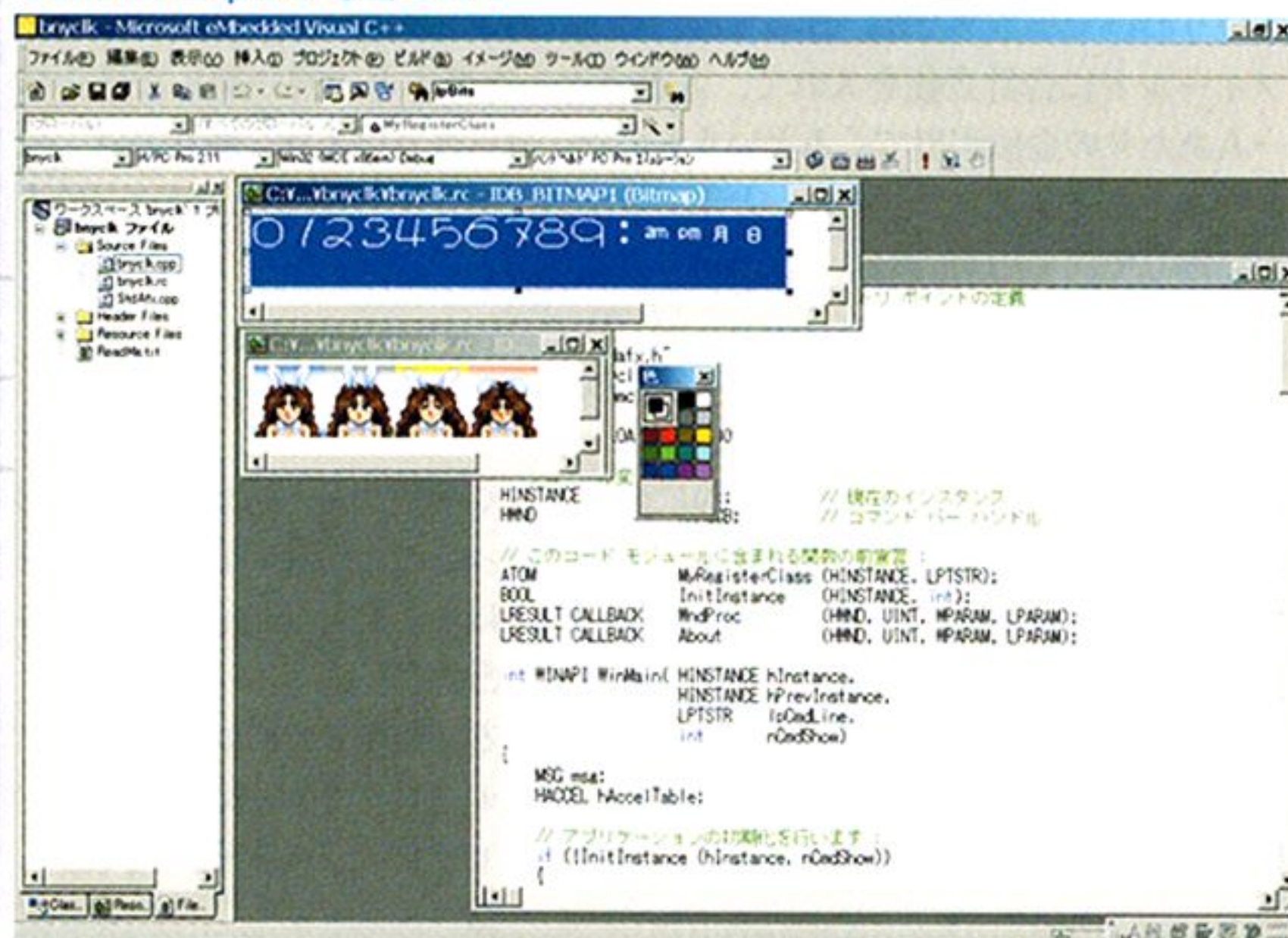
eMbedded Visual BASICでの実行

　さて，プログラムが作れたら，これのデバッグ，実行だ。

　eMbedded Visual Toolsのいいところは，作ったプログラムのデバック実行が実に簡単にできることだ。プログラマが明示的に転送などを指示する必要がない。たとえば，作ったプログラムを実行してみるには「プログラム」-「実行」メニューを選択する。これで，開発マシン上のH/PC（PocketPC）エミュレータか，あるいはActiveSync（WindowsCEマシンを，信頼関係を結んだWindowsマシンと接続してシンクロナイズさせるためのソフト）で接続されているWindowsCE機に作られたプログラム本体と，必要なdllな

図3　WindowsNT系OS上で動作するH/PC Proエミュレータ。CPUこそ「Intel」と表示されるがそれ以外は本物そっくり。ちゃんとPocketWordやInternetExplorerも動くのだ

図4　WCE MFC AppWizard (exe) を作るときにはWIndowsSocketやコマンドバーのつけ方も設定できる

図5　これこのとおり，作ったプログラムはエミュレータ上でも，そして実機上でもちゃんと動くわけだ

どがCE機の「プログラム」フォルダに一式自動で転送され，そしてプログラムも行われるのだ。

なお，Microsoft eMbedded Visual Basic 3.0は，インタプリタ型言語で，プログラムは中間言語にはなるが，実行はコマンドインタプリタが逐次ステートメントを解釈しながら実行する。そのため，eMbedded Visual Basic ではスタンドアロンの実行可能(.exe)ファイルが作成されるのではなく，中間（.vb）ファイルが作成され，そのファイルがターゲット デバイスによって実行されるようになっている。また，そのため，ランタイムライブラリ自体はターゲットとなるCPUによって違うものとなるが，プログラム自体は共通のものを使うことができる。

なお，(筆者は試していないので，マニュアルを見る限り，だが) eMbedded Visual Vasicでは，プログラム中からWindows CEのAPIを呼び出すことの可能のようだ。Windows CE APIはWin32 APIのサブセットだが，必要な機能はちゃんと揃っている。

¥Windows CE Tools¥BIN¥WINCEAPI.txt

というファイルにはAPI定義が記録されてようなので，一度そちらを確認してみるといいだろう。

WindowsCEを端的に

ここで改めて，そもそもWindowsCEとはどういうものなのか見直してみよう。このOSは32ビット版のWindowsをもとにしており組み込みシステム用の小型オペレーティングシステムだ。いろいろカスタム化が可能で，ハンドヘルドPC，PDA (PocketPC),家庭用ゲーム機，Auto PCなどいろいろなWindowsCE搭載機があるが，ユーザーが手軽に入手できてプログラミングが可能なのは主にハンドヘルドPCとPDAだろう。現在，WindowsCE搭載のハンドヘルドPCとPocketPC，WindowsではCPUとして，マシンのCPUはMIPS R4000シリーズ，INTEL Strong ARM，HITACHI SuperH (S3,S4) SHなどが使われている。近々x86を搭載したWindowsCEが発売されるという話もある。

WindowsCEのカーネルは，通常のWin32プロセスとスレッドモデルをサポートし，8レベルの優先順位を持ったラウンドロビン型スレッドスケジューリングになっている。プログラムという点で見ると，APIは基本的には，Win32 APIのサブセットとなっているので，Win32でのプログラミングをしたことのある人にはかなり楽勝感が高いはずだ。画面になにかを書くにはペイントメッセージのハンドラでBeginPaint，hDCを使って描画，EndPaintという手順もまったく同じだ。また，MFCの型名，メソッド名などもかなりの部分でWin32のそれと共通になっていて，COMを使うこともできる (ただし，非力なマシンなためか，結構オーバーヘッドが大きくなってしまうが)。

ただし，引数の型が変更になったりパラメータの種類が制限されたりしているAPIもあるので，Win32，Win16でのソースリストをそのまま持ってきてコンパイラに通るわけではなく，多少の修正は必要になるのが普通だ。たとえば，Win32APIではメモリの割り当てはGlobalAlloc，

LocalAllocがあるが，WindowsCEでは，グローバルヒープとローカルヒープの区別はないのでGlobalAllocはなくLocalAllocのみになっている。

逆にWindowsCEでは，たとえばWindowsにはないCommandBar(画面上部のメニューやボタンが乗るバーのこと)があるため，APIにもCommandBar関連などのWIN32にはないものが新しく追加されている。あるいはデスクトップWindows機から制御されるためのRAPI(Remote API)などもそうだ。

eMbedded Visual C++ 3.0でのプログラミング

さて，eMbedded Visual C++3.0だ。これはeMbedded Visual Toolsに含まれるCコンパイラで，扱うのがWindowsCEであるということ以外は見かけも機能もほとんど，Windows用のプログラミング経験のある方なら一度は使ったことがあるだろうVisual Studio(Visual C++)そのものだ。本格的にWindows用のプログラミングをしたことがあるなら，一度はVisual Studio(Visual C++)のお世話になっていることだろう。

このeMbedded Visual C++もeMbedded Visual BASIC同様，WIN32からのサブセット版であるものの，プログラミングの手順，操作方法などはそれらと同じになっていて，感覚的な違和感はないはずだ。MFCやATLも利用可能になっていて，プログラム作成を手伝ってくれる。ただし，残念ながらSTLは使用できない。

この開発ツールで作ることができるオブジェクトはMIPS，ARM，SH3，SH4，x86，それとx86の(WindowsNT系OS上で動いている)エミュレータ用のそれぞれdebug，release版だ。VisualC++は，CPUネイティブのコードを作るので，当然，ターゲットCPUの指定が必要になるわけだ。デバック情報がバイナリに含まれているのがdebug版，そうでないのがrelease版なのも同じだ。

プログラムをコンパイルして，デバック実行，あるいは実行すると，eMbedded Visual Basicの場合と同様にエミュレータ，あるいは接続されたWindowsCE機にプログラムが転送されて，実行される。実機でもステップ実行が可能だ。

それでは，実際にプログラムを作ってみよう。

eMbedded Visual C++では，プロジェクトを新規に作成する場合，次のようなプロジェクトのなかから新しく作るものを選ぶことができる。

- WCE Application
- WCE ATL COM Appwizard
- WCE Dynamic-Link Library
- WCE MCF ActiveX Control
- WCE MFC AppWizard (dll)
- WCE MFC AppWizard (exe)
- WCE Pocket PC Application
- WCE Pocket PC MFC Application
- WCE Static Library

「WCE Application」は簡単にいうとCで書かれたトラディショナルなスケルトンを作ってくれる。使用するAPIはSDKレベルのものだ。「WCE MFC AppWizard (exe)」はアプリケーションウィザードを使ってC++で書かれ，MFCを利用したもの。「WCE Pocket PC Application」「WCE Pocket PC MFC Application」はそれぞれのPocket PC用のものを作る。あとは，「WCE ATL COM Appwizard」がATLを利用する，それ以外はライブラリを作るときなどに使うメニューだ。

たとえば，WCE Applicationを選んでみよう。プロジェクトの種類としては「空のプロジェクト」「単純なWindowsCEアプリケーション」「標準的なHello! Worldアプリケーション」のどれかを選んで作ることができる。基本的に最小限だが，アプリケーションプログラムに必要な部分はスケルトンとしてひととおり作ってくれるので，ここでは「Hello! World」を選ぶのが便利だ。これで「終了」をクリックするとリソース，ソース，ヘッダファイルのひととおり入ったスケルトンのプロジェクトが作られる。

ここで，メインのソースになる.cppファイルを開いてみてほしい。たとえば，プロジェクト名がtestであるならば「test.cpp」という名前になっているはずだ。

リスト

```
Option Explicit

Private Sub Command1_Click()
    Text1.Text = Int(Text1.Text) * 1.05
    Command11.Caption = Text1.Text
End Sub

Private Sub Command10_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 10
End Sub

Private Sub Command11_Click()
    Text1.Text = Command11.Caption
End Sub

Private Sub Command12_Click()
    Command11.Caption = 0
End Sub

Private Sub Command2_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 2
End Sub

Private Sub Command3_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 3
End Sub

Private Sub Command4_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 4

End Sub

Private Sub Command5_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 5
End Sub

Private Sub Command6_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 6

End Sub

Private Sub Command7_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 7
End Sub

Private Sub Command8_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 8
End Sub

Private Sub Command9_Click()
    If (Not Command11.Caption) Then Command11.Caption = Text1.Text
    Text1.Text = Int(Text1.Text) / 9
End Sub


Private Sub Text1_Change()

End Sub
```

これが，int WINAPI WinMain(……を使ったSDKレベルのWindowsプログラムになっているのがわかるだろう。

WndProc()内にこのプログラムのイベントハンドラとなるプログラムが書かれている。たとえば，case WM_PAINT:～breakの部分に，このプログラムのメインウィンドウにPAINTメッセージが届いたときの処理の方法が書かれているわけだ。ほかのたとえば，ここにダブルタップされたときの処理を追加したいときには，

```
case WM_LBUTTONDBLCLK:
            :
            :
(ダブルタップされたときの処理)
            :
            :
```

図6　eMbedded Visual C++ での開発画面。基本的に操作感覚はVisual C++とまったく同じ。AppWizでスケルトンを作り，ClassWizでクラスを増やして実装していくのも同じだ

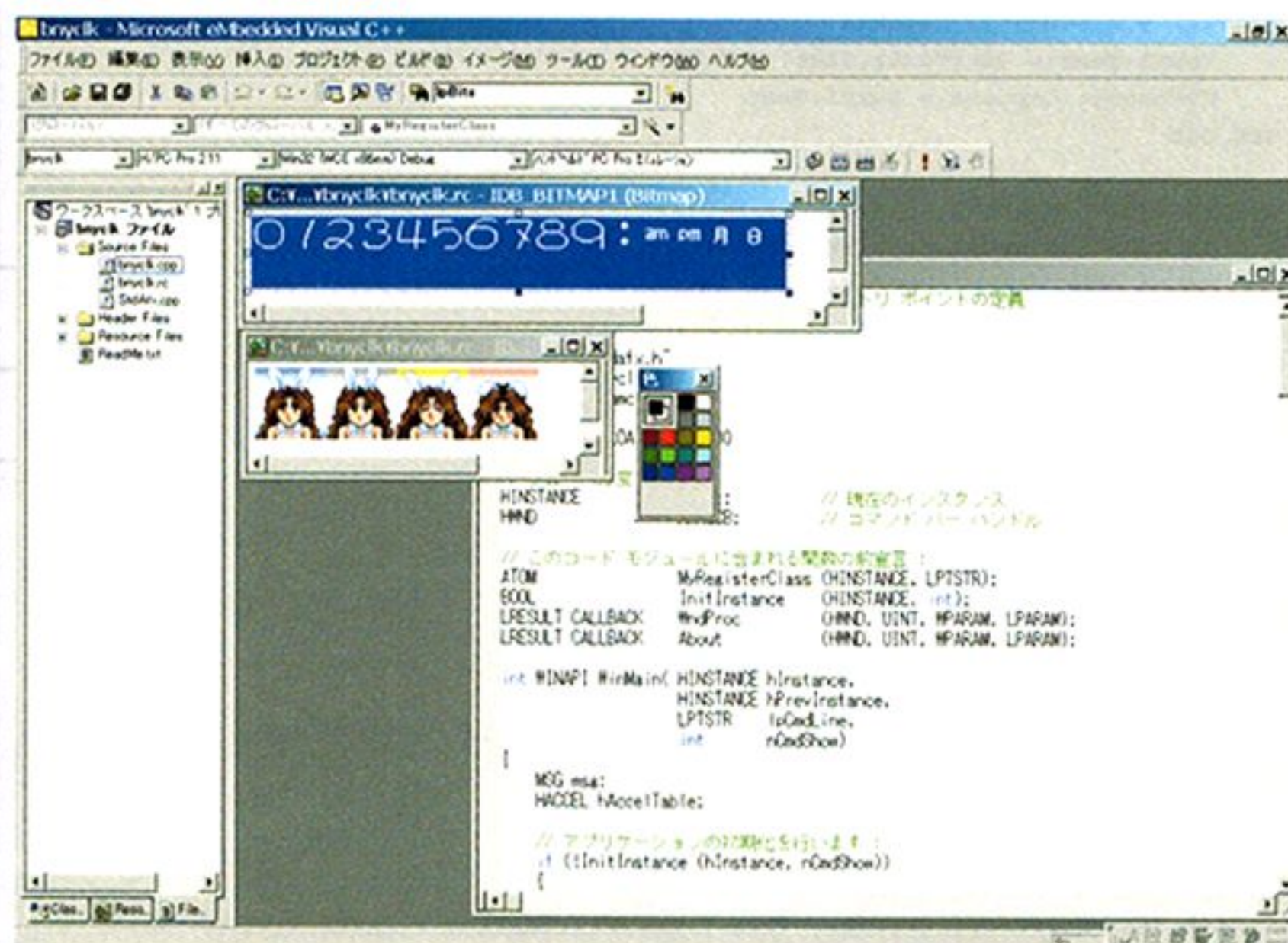
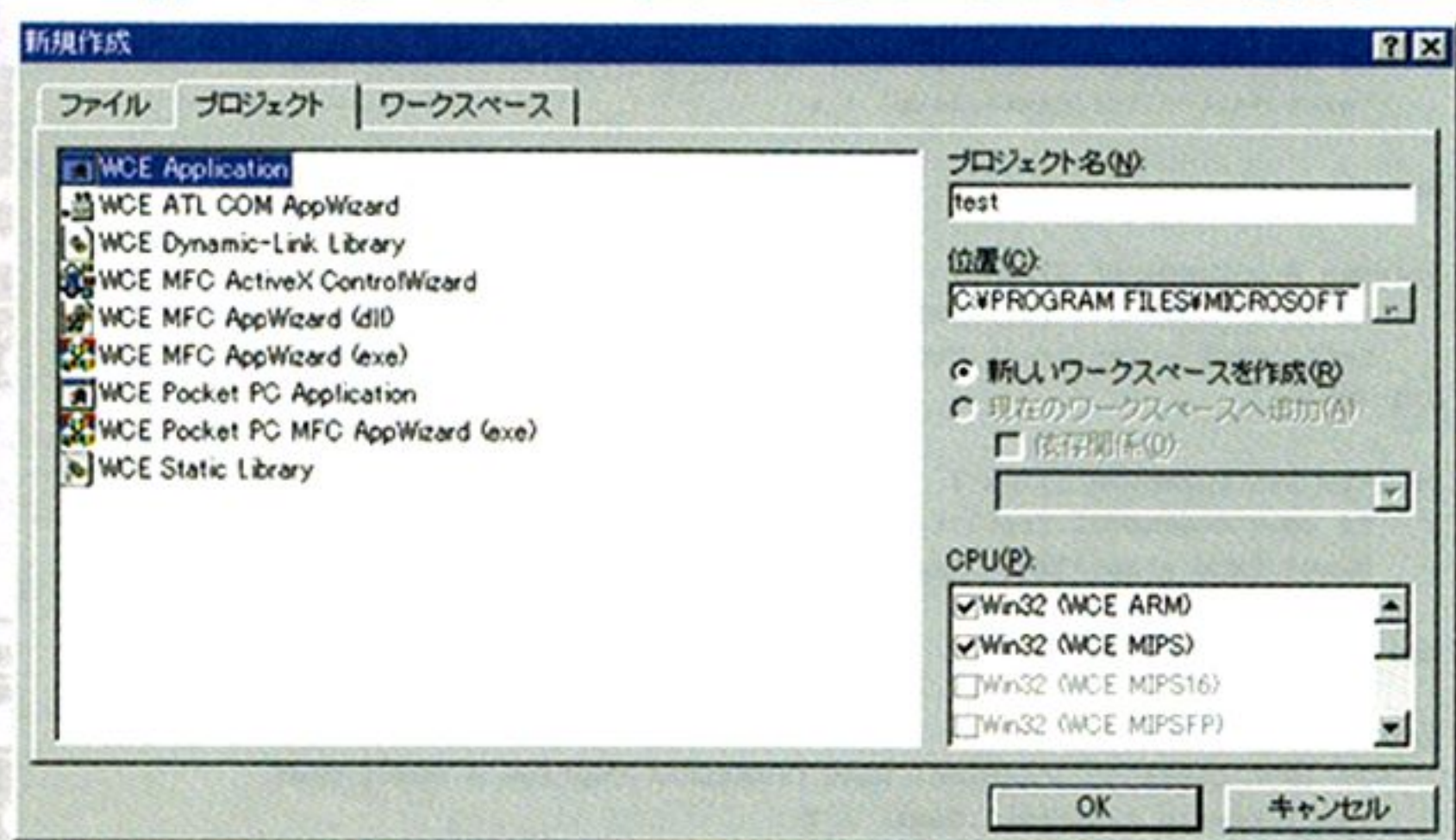



図7　AppWiz画面。ターゲットと，プロジェクトのタイプをここで決める

```
break;
```

と追加すればいいわけだ。またAbout(　)のほうにはバージョン情報を表示するダイアログボックスのメッセージ ハンドラがあるので，ほかにも表示したいダイアログを作る場合はここを参照して作ればいいわけだ。

この辺の手順はWIN32,WIN16プログラミングと同様なので，APIの名前や引数などに違いがあることに気をつければWindows9xやNTからのプログラム移植作業自体もそう難しくないだろう（ただし，画面周りや操作性の違いには十分気をつけたほうがいいが。WindowsCEにはウィンドウは基本的にサイズ変更しないし，常に最大化した状態にすることが推奨されている。マウスカーソルや右クリックも存在しないし，そもそも画面がかなり小さい）。

また，アプリケーションウィザードを使ってプログラムを作る場合だが，WCEアプリケーションの作成と同様に，プロジェクトを新規に作成する際に，

・WCE MFC AppWizard (exe)

を選ぶ。続いては，アプリケーションウィザードでアプリケーションの種類を「SDI」，「ダイアログベース」のいずれかから選ぶことになる。なお，PocketPCの場合は「ドキュメントリストのあるSDI」という項目も加わる。この項目は，たとえば，PocketPCのPocketWordなどを起動してみてほしい。起動すると同時にファイルリストが表示されるはずだ。あれをプログラムの起動と同時に表示してくれるスケルトンを作るのが「ドキュメントリストのあるSDI」である。ちなみに，WindowsCEには「MDI」という概念はない。

また，ここで「ドキュメントービューアーキテクチャのサポート」を選ぶとWIN32プログラミングの場合と同様なドキュメントビュータイプのスケルトンを作成する。また，このアプリケーションウィザードではアプリケーションがWindowsソケット，Windowsヘルプ，ActiveXコントロールを使用するかどうか，コマンドバーのタイプを選ぶことができる。これでスケルトンが作成できたら，クラスウィザードなどを使ってプログラムを作り込んでいけばいいわけだ。

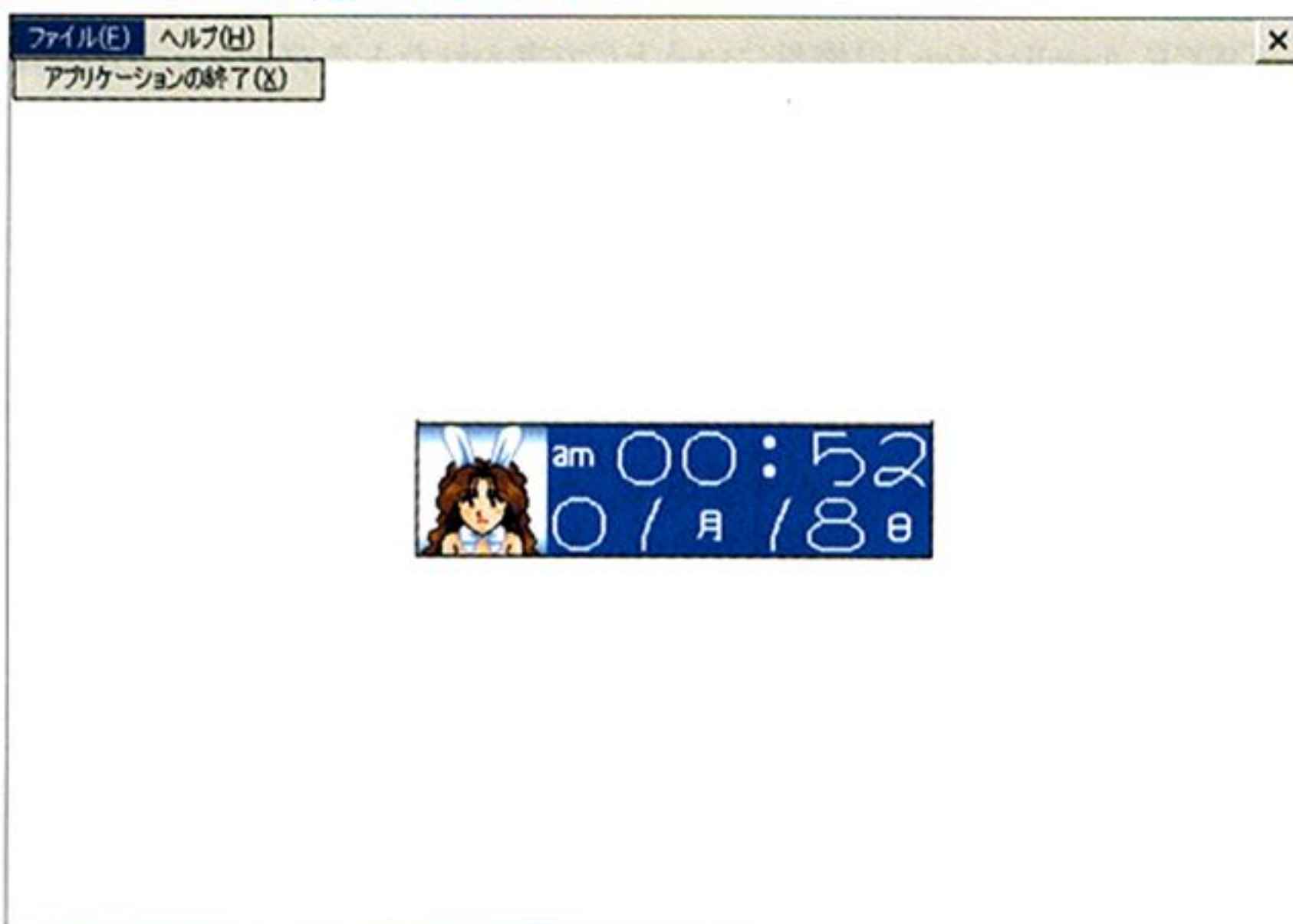
図8　eMbedded Visual Basicでの開発画面。フォームにコントロールを載せて，それぞれのコントロールへにイベントが発生したときの挙動をBASICで描くだけでH/PCやPoketPCで動くプログラムが作れる。実にお手軽

最後に

さて，WindowsCEプログラミングは，Windowsプログラマにとってはかなり面白い環境だ。なにしろWindowsプログラミングの勘所の多くが通用するし，それでいてまだWIN32ほど，なんでもアプリケーションが揃っているわけはない。そのうえ開発に必要なものはこのソフトとあとはせいぜいH/PC，PocketPC本体くらいなのだ。

さて，Windowsプログラミングと比較して困る点は，シリアルポートだと，作成したプログラムの転送が終わり，実機でプログラムをするのに時間がかかる点だろうか。これに関しては，できれば，一度信頼関係を結んでしまったら，あとはEtherなどを使ってCE機と母艦をLAN接続してしまえば楽だ。PC Cardだけでなく，CFカードスロット用のLANカードなども最近では発売されている（早くSDカードなどでも発売されるといいのだが……）。

それと，プログラミングするときにはCE機は必ずACアダプタに接続しておいたほうがいい。なんだかんだでプログラミングというのは結構電池を食うのだ。

図9　バニーさん時計のできあがり

WonderSwanでゲームを作ろう

Yamato Satoshi 大和　哲

携帯ゲーム機のみならず市販ゲーム機のなかで唯一，開発環境がユーザーレベルにまで公開されているのがバンダイのWonderSwanです。WonderWitchを使えばC言語レベルでさまざまなアプリをプログラミングできます。ここではWonderSwanの概略と開発の手順を紹介します。

バンダイから発売されている携帯ゲーム機「WonderSwan」はQuteから発売されている「WonderWitch」という開発キットを使うことで，ユーザーレベルでプログラミングを行うことができます。このWonderWitchには，WonderSwan用の専用カートリッジ，PCとWonderSwanの接続ケーブル，それにCD-ROMにマニュアルが含まれています。作ったプログラムはこのCD-ROMに含まれるLSI-C 86 for WonderWitch，あるいはTurboC 1.0を使ってコンパイルします。つまり，プログラムを作成するためには主にCコンパイラが使われることになるわけです（当然，アセンブラの使用も可能です）。そして，作られたプログラムは，不揮発メモリを搭載したWonderWitchカートリッジを挿入したWonderSwanに転送して使うことになります。

この記事では，WonderWitchを使ったプログラミングの入門として，WonderSwanのアーキテクチャの簡単な解説と，WonderWitchの簡単な解説，そして極簡単なゲームを実際にプログラミングする様子を解説します。

なお，すでにカラー液晶を搭載した「WonderSwan Color」も発売されていますが，これの発売と同時にWonderWitchにもカラー対応ライブラリがリリースされる予定です（ライブラリ，とはいってもCのライブラリだけでなく，おそらくビットマップ変換ツールのカラー対応かなどもあるでしょうが）。WonderWitchそのものには変更はありません。

さて，WonderSwanのアーキテクチャですが，ひと言でいうと「昔の2Dゲーム世代のハードウェアをごく簡単にしたようなもの」という印象のものになっています。

・CPU

CPUは使われるコンパイラがLSI-Cであることからもわかるようにx86互換の16ビットCPUです（より正確にはV30MZをコアにしたカスタムCPUであるといわれています）。なお，LSI-Cではオブジェクトは基本的に86用のコードが出力されいます，186互換であるV30MZ特有の命令を使ったオブジェクトは出力されていません。また，186最適化オプションのようなものもありません。

・メモリ

これはWonderSwanではなくWonderWitch側の仕組みなのですが，ユーザーエリアとして使用できるのは256KBのSRAMです。64KBずつ４つの分かれていて，それぞれ，オペレーティングシステム用ワークエリア，ユーザープロセス用データエリア×2，ソフトファイルシステムとなっています。また，システムとユーザープログラムのスタック用として16KBのメモリがWonderSwan本体搭載されています。

WonderSwanのCPUは186互換ですので，x86系CPUにはつきもののセグメントをプログラミングの際には考慮しなければなりません。WonderWitchでは基本的にプログラムはスモールモデルでの利用になります（そもそもLSI-C for WonderWitchではスモールモデルしか利用できないようになっていますが）。ですので，最大でもコードサイズ64KB・データサイズ64KBが使用可能領域ということになり，デフォルトのポインタはnear型，宣言された変数はひとつのセグメント（データセグメント）に置かれるということになります。WonderWitchのマニュアルの範囲内で使うのであればそういうことはまずありませんが，もし，セグメントのまたいでメモリアクセスを行う場合はポインタを作るときには，

```
void far *lpmem;
```

というようにfar宣言が必要になります。

・画面周り

画面周りは，ハードウェア的には解像度が224×144ドットで白黒16階調から8階調分を選んで出力できるようになっています。アーキテクチャ的にはバックグラウンド用の2スクリーンとスプライトが重ね合わせでき，これの合成されたものが画面表示出力となります。

スクリーンに描くことのできるのはキャラクタといわれる8×8ドットのオブジェクトで，これには画面の8階調からさらに4階調分のカラーをつけることができるようになっています。つまり，個数的には224×144ドットのスクリーンですから28×18＝504個のキャラクタをひとつのスクリーンに敷きつめられることになります。ちなみにこのキャラクタは全部で512個定義することができます。

8×8ドット固定座標に敷きつめられるスクリーンに対して，好きな座標キャラクタを置いて，ゲーム中のキャラクタ表示などに使われるのがスプライトです。スプライトも大きさは1個で8×8ドット，最大128個まで定義して画面上に表示することができます（ただし，ハードウェアの制約で同一水平線上には最大32個までしか表示されません）。

また，これら，スプライト，スクリーンをどのように重ねあせるのかもソフトウェアから設定することができるようになっています。たとえば，スクリーン2枚に上にスプライトを表示したり，あるいはスクリーンとスクリーンの間にスプライトを置くこともできます。つまり，2重スクロールする背景の中をキャラクタを動かしたり，あるいはのぞき窓のような中をスクロールする背景とキャラクタ，などという表示も簡単に作ることができるわけですね。

・サウンド

翻ってサウンド方面を見てみると，WonderSwanの音声出力はサンプリングボイスも使える4チャネル構成のPCM音源になっています。これを使うには6レベル×32ステップの波形データをは登録しておき，周波数，それにボリュームを指定することで4音までの音を発生させます。それぞれのチャネルには，この通常のサウンド機能のほかに，たとえば2チャンネルの8ビットサンプリング音声の出力など特殊機能もあります。

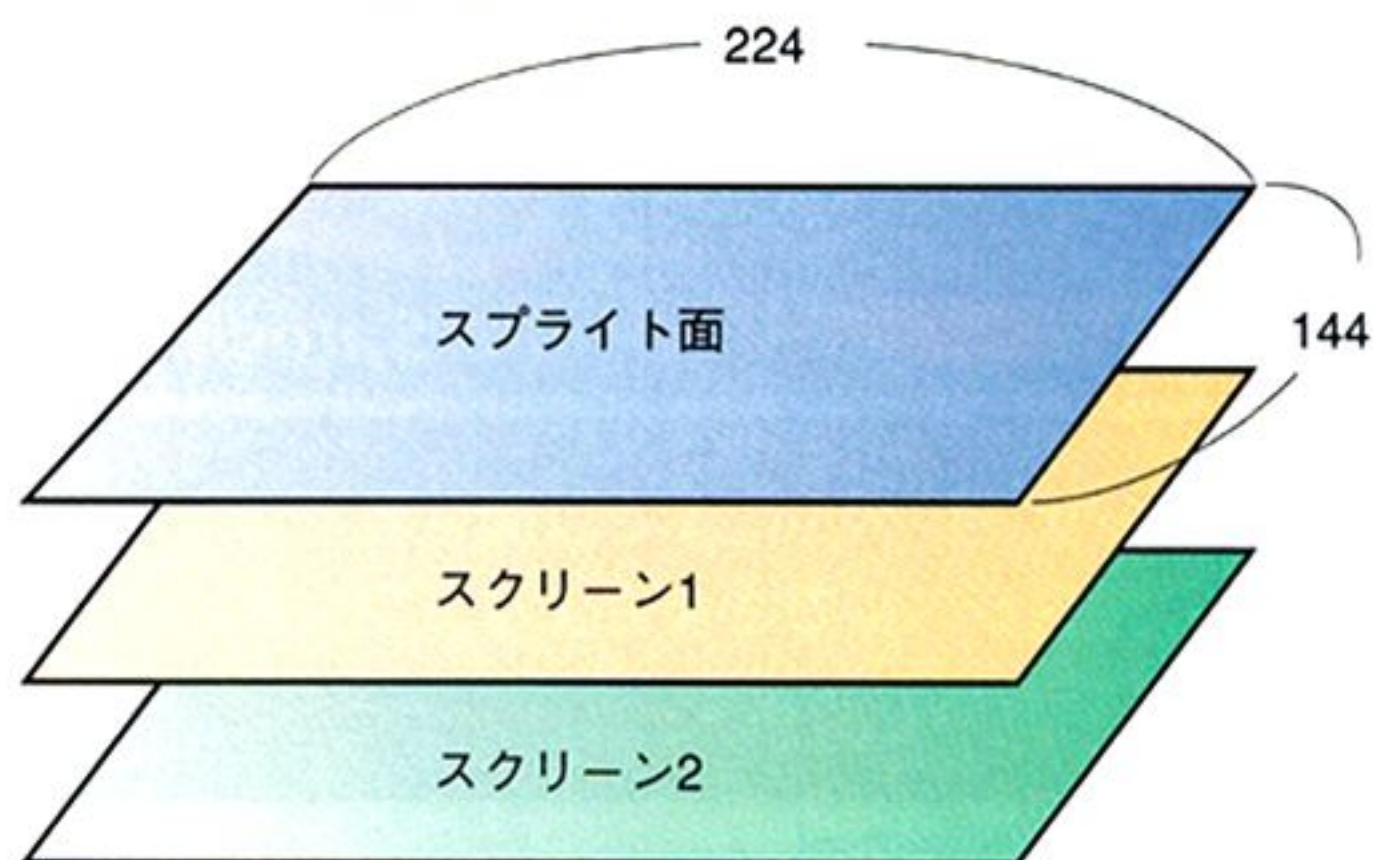
・その他のハードウェア

また，ほかには上下左右方向の4ボタンが2組，aボタン，bボタンの入力ボタン，モバイルワンダーゲートなどオプション機器の接続することのできる拡張コネクタがある。

この拡張コネクタは全2重対応の，調歩同期式のシリアルポートになっていて，仕様は以下のようになっています。

項目	仕様
プロトコル(データリンク下層)	調歩同期，全2重通信
ボーレート	9600bps/38400bps切り替え
データビット数	8ビット固定 パリティなし
送信ストップビット長	1ビット
送信バッファ	なし
受信バッファ	1段
エラー検出	受信オーバーラン
モデム制御信号	なし

図1　WonderSwanの画面構成

図2　TrausMagieでWonderSwanにデータを転送

Hello Worldしよう（プログラミングの手順）

さて，それでは実際にWonderWitch用のプログラムを作ってみましょう。プログラム作成の手順ですが，要するに簡単なクロス開発になっていて，DOS/Windows上でプログラムを作り，ここからWonderSwanにプログラムを転送します。ここでは基本的にWindows98（またはMe）でプログラミングする場合を考えてセットアップの工程を見ていきましょう。

・WonderWitch添付ソフトのインストールと環境を使うための準備

1）WonderWitch添付のファイルはCD-ROM中のインストーラを使って，ハードディスクにコピーします。

そして，WonderWitchで使うコンパイラは基本的にDOS上のものですからDOSプロンプトを起動します。DOSモードでは，コマンドはカレントディレクトリ（現在いるディレクトリ）にあるか，Pathが通っていないと動きませんのでPathの設定などが必要になります。そのために必要なバッチファイルが¥WWitchディレクトリのBINにあるSETUP.BATです。

```
C> C:¥WWitch¥Ssetup.bat
```

などとして環境をセットアップしてください。

なお，このWonderWitchでは，プログラムのコンパイルにLSI-C86 fopr WonderWitch，あるいはTurob-C 1.0（英語版）のどちらかを使うことができるのですが，TurboCコンパイラを利用する場合はTurboCのインストールも必要になります（LSI-Cの場合はインストーラを利用時にファイルが展開されているので必要ありません）。

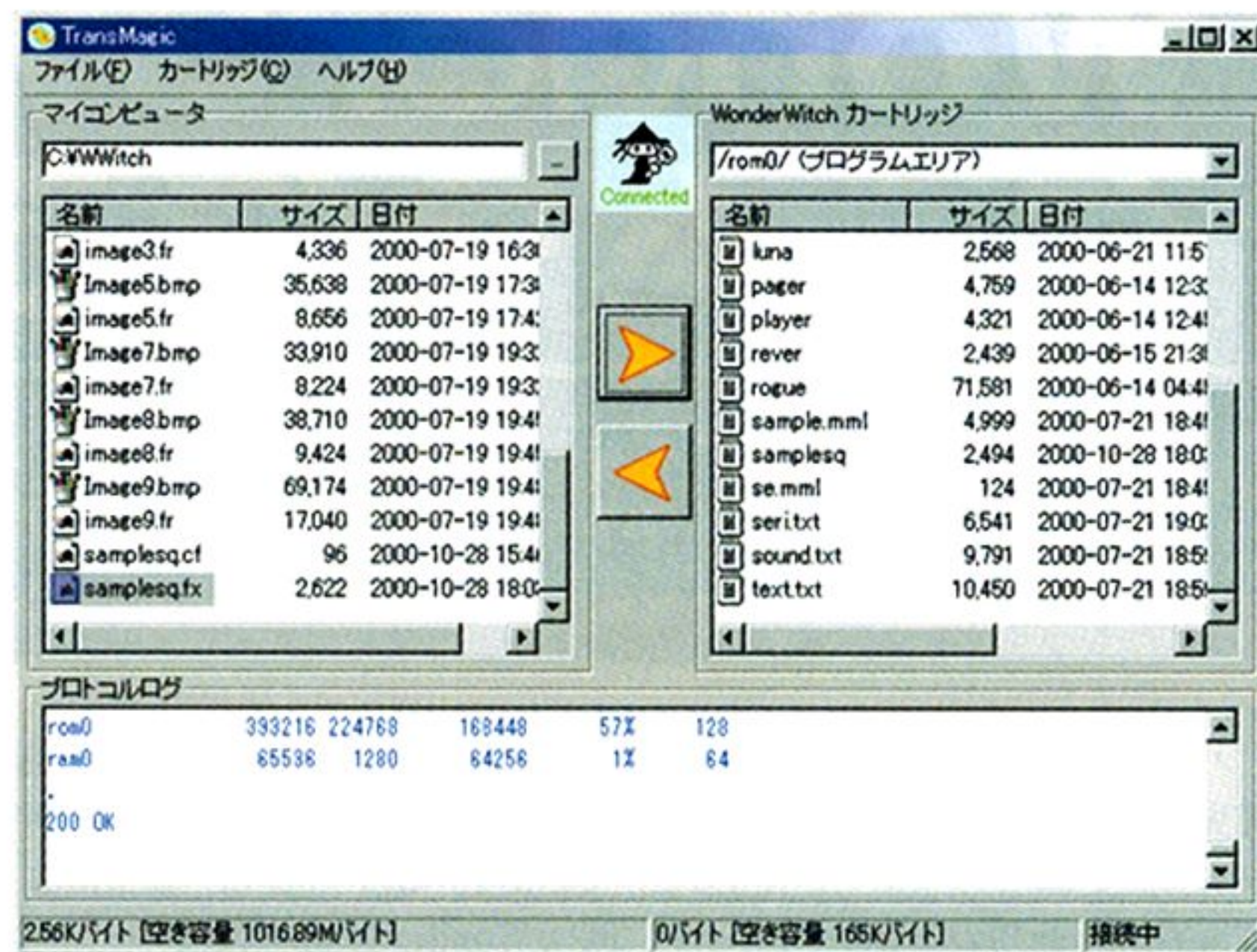
なお，このページでは，以降，LSI-Cを利用するものとして解説しています。サンプルもすべてLSI-Cでコンパイルしています。

・プログラミング

2）DOS，あるいはWindows上でC言語のソースリストと，WonderWitchバイナリを作るための.cfファイルを作成します。どちらもテキストファイルですので，Windows上のテキストエディタなどで作成しましょう。ファイル名は，拡張子を.Cとして，

```
ファイル名.C
ファイル名.CF
```

として保存しておきます。

.cfファイルは最終的にどのようなWonderWitchバイナリを作るかという指示が書かれています。WonderWitchプラグラミングでは，PC上のコンバータにて，Cコンパイラで作ったオブジェクトファイルをWonderWitchバイナリ形式に変換することになるのですが，このときに，プログラムの説明文などWonderWitchプログラム特有の情報がバイナリに付加されます。この情報を，たとえば，

```
name: Squash
info; スカッシュゲーム
```

というようにこの設定ファイルに書き込んでおくわけです。

3）Cソースをコンパイル

LSI-C，あるいはTurbo-CでCソースをコンパイルします。通常，DOSなどでCコンパイラでプログラミングするときは，ターゲットの実行ファイルまで一気に作ってしまうのが普通ですが，WonderWitchでは，Cコンパイラ.exeファイルの一段階前のオブジェクト（.obj）ファイルを作ります。というのもファイルコンバータのmkfentはこのオブジェクトファイルをもとにWonderWitch用バイナリを作るからです。ですので，コンパイルを行うときはDOSプロンプトで，

```
lcc86 -o ファイル名.bin ファイル名.c
```

というように行います。すると，

```
ファイル名.bin
```

という名前のオブジェクトファイルができあがります。

4）WonderWitchバイナリの作成

続いては，コンバータを使って，オブジェクトファイルからWonderSwanバイナリを作ります。このmkfentはすでに作られている設定ファイルをもとにWonderWitchバイナリを作りますので，実行にはDOSプロンプトで，

```
mkfent ファイル名.cf
```

と入力するだけです。

なにもエラーなどの問題が起きなかった場合，カレントディレクトリには

```
ファイル名.fx
```

という名前のWonderWitch用バイナリファイルができあがっているはずです。

5）WonderSwanに転送する

あとは，できあがったWonderWitchバイナリをWonderSwanに転送すればOKです。付属のコード（2本を接続し，PCのシリアルポートとWonderSwanの拡張コネクタに）差し込みます。WonderSwan側は

WodnerWitchカートリッジを挿入して電源を入れておきます。特にWonderWitchカートリッジのシェルをMegからほかのシェルに入れ替えなどしていない場合は，ここでMegの「ファイル転送」機能を使ってWonderSwanにファイルを転送すれば作業は完了，ということになります。

WonderSwanのハードウェアアーキテクチャの概略

さて，続いては，最初にWonderWitchのプログラミングに必要な，WonderSwanのアーキテクチャの解説です。

WonderWitchでは，ハードウェアを直接叩くことはないものの，それに密着したBIOSを利用するプログラミングがメインになりますので，活用するためにはある程度ハードウェアやそれに近いソフトウェア（BIOS）に関する知識があるほうが便利です。逆にいうと，WonderWitchは使い方次第ではWonderSwanの機能のかなり深い部分まで利用したプログラミングができるツール（というかそれがある程度必須ともいっていいくらいの）ツールでもあります。

ゲームを作りたい，あるいはWonderSwanの機能をできるだけ使ったプログラムを作るということになると，WonderSwanのスペックを知っておくことは必須になるでしょう。ここにWonderSwanのハードウェアの主な仕様を挙げておきます。そして，サンプルとして，WonderSwanの「スプライト」を使った簡単なゲームを作ってみましょう。

・グラフィック周り

古くからX68000などでのプロミングをされている方にひと言でわかるようにいってしまうと「8×8ドット単位の『スプライトバックグランド』構成で最大解像度ははは224×144ドット，キャラクタは最大512個まで登録可能，スプライトは水平32個まで表示可能」ということになります。

これを1つひとつ解説していくと，まず，WonderSwanでは画面に表示する絵は8×8ドットの単位がひとつになっていて，1つひとつが「キャラクタ」と呼ばれています。このキャラクタは好きなパターンで定義することができます。

そして，WonderSwanの画面は内部で3層の構造になっていて，ひとつがスプライト画面，ほかはスクリーン1，スクリーン2と呼ばれます。スプライト画面には，先ほどのキャラクタのデータを画面の好きな位置に配置することができます。つまり，画面上を動くキャラクタなどはこれを利用して表示すればいいわけですが，横には最高でも32個までしか並べることができず，それ以上並べた場合にはそのスプライトは表示されません。

そして，スクリーン1，2には先ほどのキャラクタを敷きつめることができます。座標は8×8ドット単位に固定されてしまいますが，その代わり個数に制限なくいくつでもびっしり並べることができます。なお，スプライトは同じスプライトを座標を変えて表示すると前にいた座標の自分は消えて新しい座標に表示される（ひと言でいうなら「移動する」）わけですが，キャラクタは新しい座標に表示しても古い座標に表示したものは消えません。つまり，同じキャラクタを敷きつめてゲームの背景などに使うことができるわけです。

ちなみに，設定によってどのプレーン（スクリーン，スプライト）を使用するのか，あるいはどのような優先順位にするのかも設定可能で，たとえば，スクリーン1を前景，スプライトをキャラクタ，スクリーン2を背景に使うこともできますし，そうではなく，スクリーン1，2とも背景に使って2重スクロールさせる，ということもできます。

なお，スクリーンは224×144ドットですからキャラクタに直すと縦28個，横18個。個数にすると504個ですから，スクリーンにキャラクタひとつずつ敷きつめて，1つひとつ書き換えていけば，これで擬似的にグラフィック画面のように使うこともできます。

また，この画面の色数なのですが，モノラルのWonderSwanの場合，画面は白黒16階調となっています。ただし，ハードウェアの都合で画面に実際に表示できる階調は16階調のうちの8階調分のみ。さらにキャラクタに関してはこの8階調のうち，4階調しか使用することはできません。

図3 WonderSwan側の通信画面

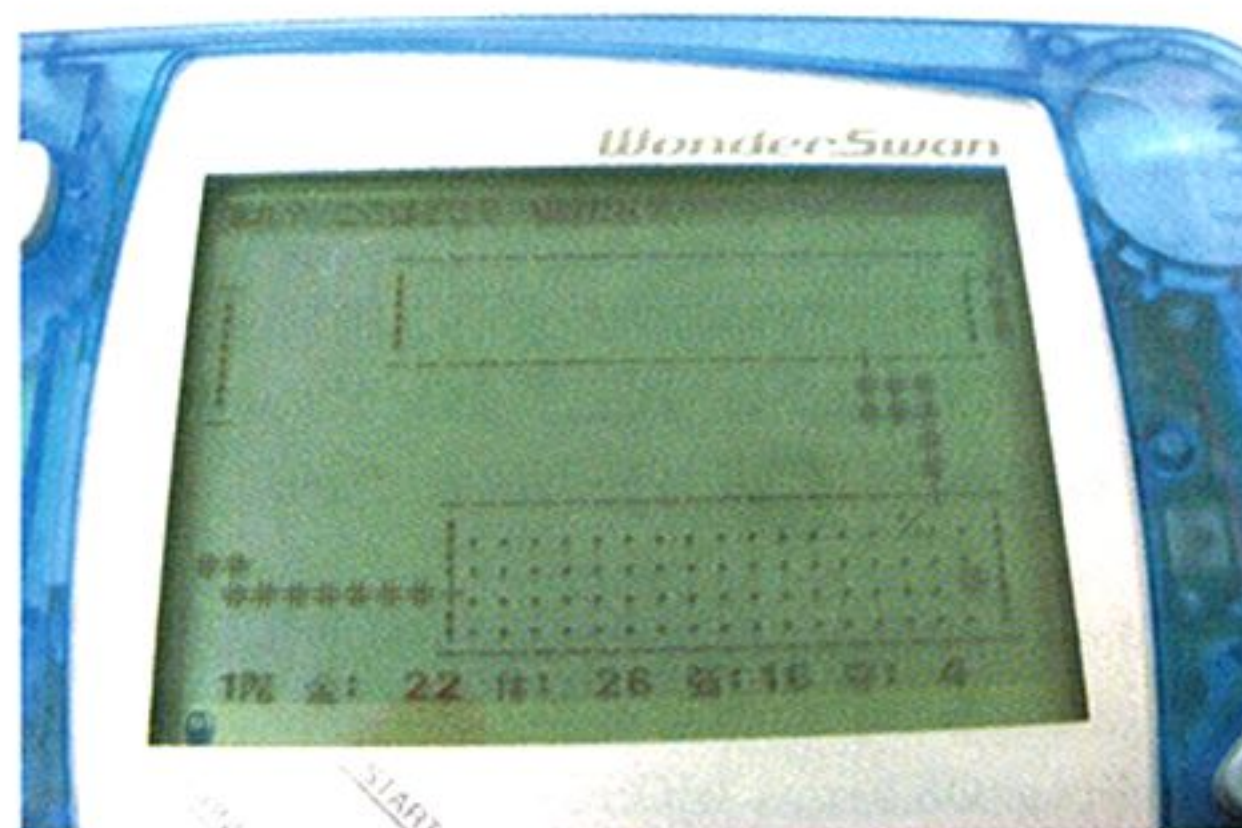



図4 WonderWitchのサンプルで入っているRogue Clone

図5 WonderWitch専用のデータカートリッジ（3980円）

・サウンド周り

WonderSwanのサウンド機能としてはは，4チャネル構成のPCM音源ユニットが搭載されています。

登録された16レベル×32ステップの波形データと周波数，それにボリュームを掛け合わせたデータをレジスタに登録することで，音を出すことができます（なお，前述のように，SoundILを使ってプログラミングすればレジスタを直接叩かずに，MMLデータを再生させることもできます。音楽演奏などの場合はこの機能を使うほうがかなり便利です）。PCM音源というよりは波形メモリタイプの音源です。

ちなみに，このサウンドLSIのチャンネル1つひとつには基本となる通常の4ビットPCM機能のほかに，いくつかのチャネルに特殊機能がついています。チャネル2が8ビットのサンプリングデータをそのまま再生するボイスモード，チャネル3はスイープ，チャネル4はノイズ音を出すことができます。

・その他の機能

WonderSwanにはほかにも，拡張ポートと，ボタンがあります。

拡張ポートはWonderWitchから見ると純粋なシリアルポートとして見えます（というか，実際，ハードウェア的にも単なるシリアルポートなのです

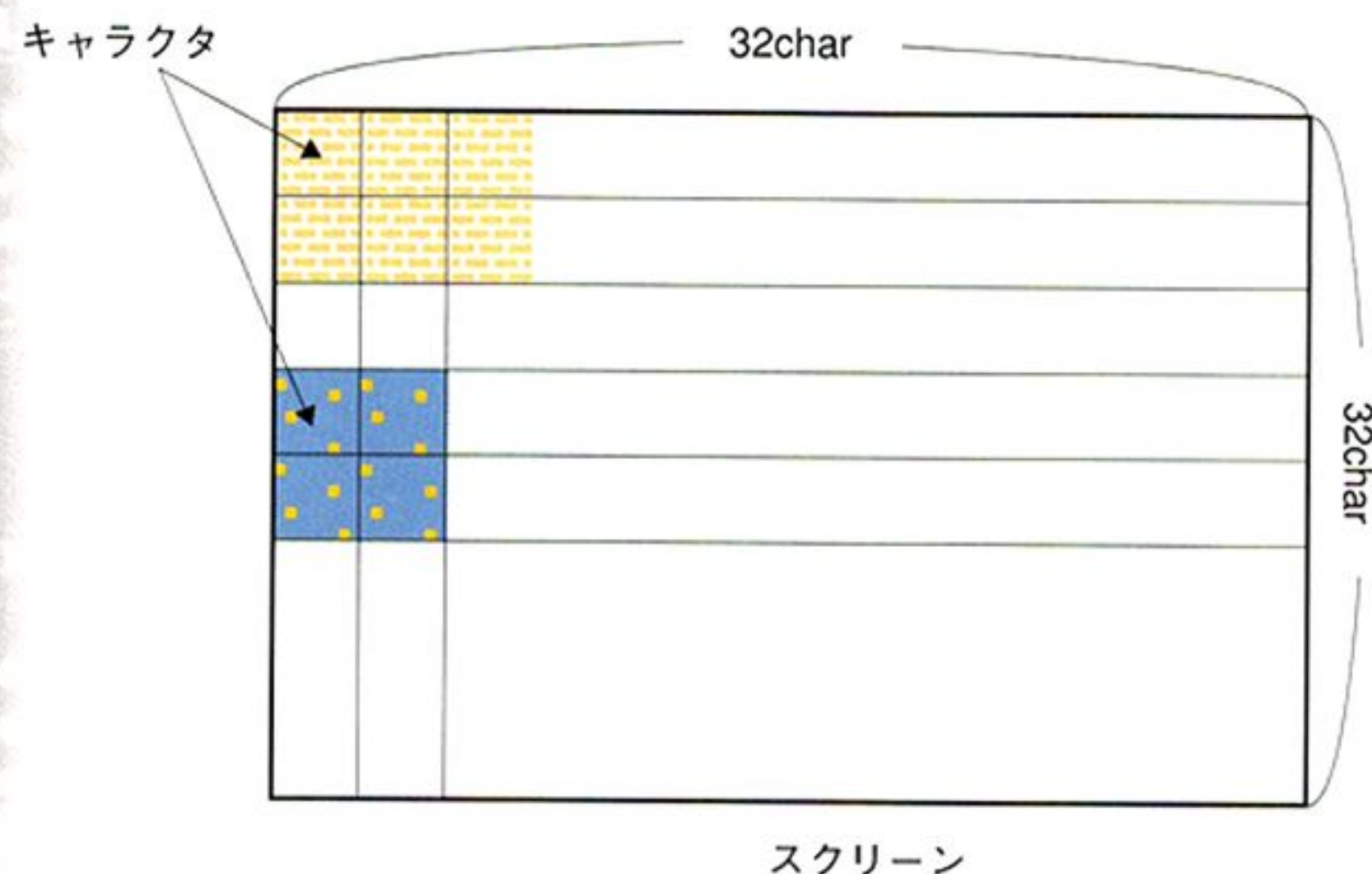
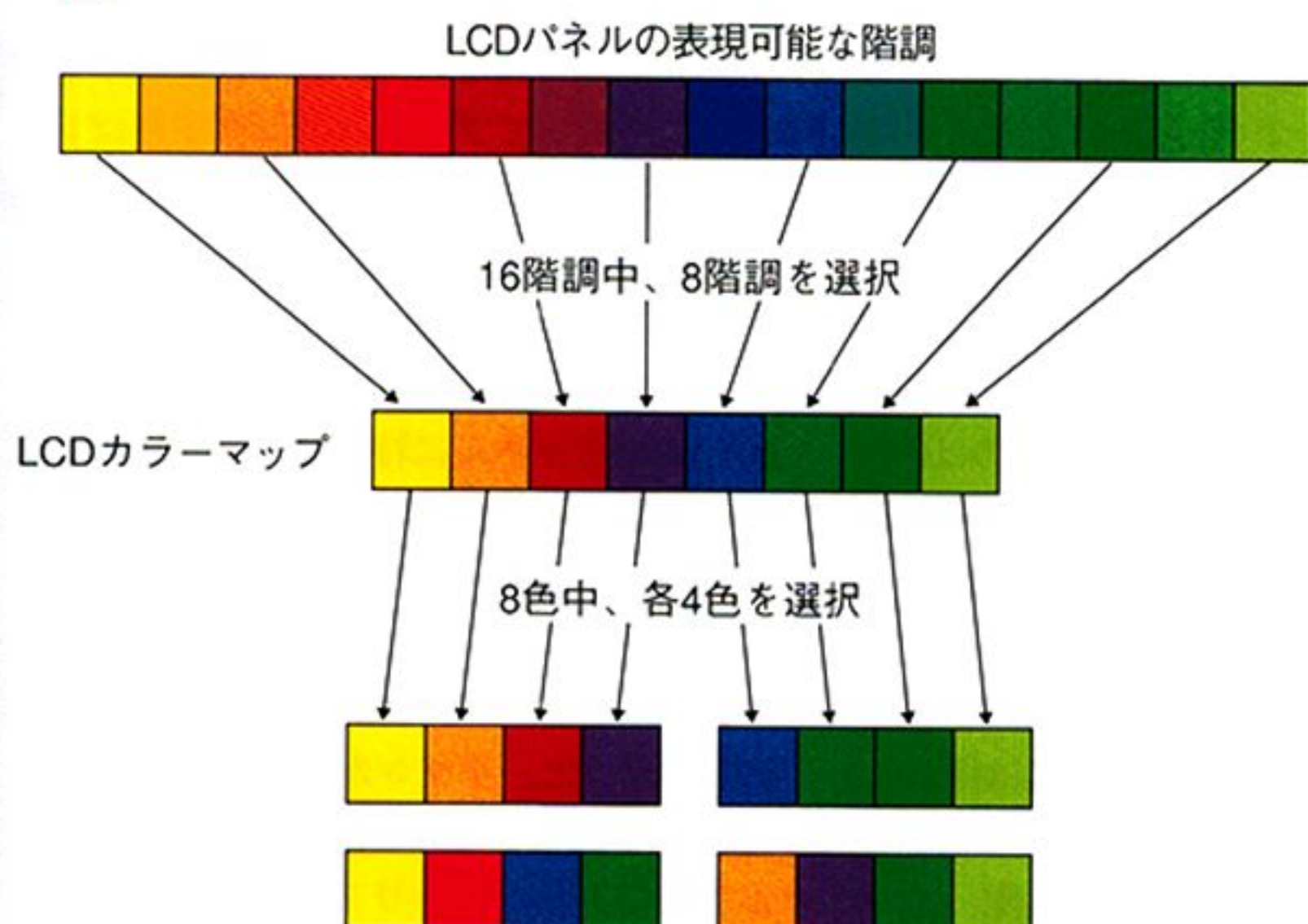
が……)。ライブラリもシリアルポートのオープン，クローズといったファンクションコールが用意されているので，これを使ってプログラミングすることになります。

また，WonderSwanには4方向ボタンが2つと2方向ボタンが2つあります。ハードウェア的には割り込みでボタンが押されたときのその立ち上がり立ち下りが割り込みで知らされるようになっていて，同時押しの判定が可能になっています。ただ，WonderWitchでは，ボタンが押された情報を得るためには特に割り込みハンドラを用意する必要はありません。WonderWitchカートリッジ上のFreyaBIOSがラッパーになっていて，関数を呼び出すことで簡単にどのボタンが押されているかを知ることができるためです。

WonderSwanの機能をBIOS経由で利用する

さて，WonderWitcを使ったhプログラミングでは，これらハードウェアをBIOSコールして使うことになります。使われているCPUがV30MZですから，メモリ保護などは行われていません。ですので，直にメモリやI/Oを叩くこと自体はできなくはないのですが，WonderWitchではWonderSwanのハードウェアの具体的な操作方法(たとえば，メモリマップやI/Oポートの具体的な番地)は記載されていませんので，これがWonderSwanの機能を使うにはこれがほぼ唯一の方法，ということになります。

このBIOSはWonderWitchカートリッジに搭載されている不揮発メモリ内にあり，Freya BIOSと呼ばれています。このWonderWitchを使ったプログラミングはFreyaOS,Freya BIOSを利用するために，WonderWitchカートリッジが必須となります。

Freya BIOSには，スクリーンへの文字列出力，スクリーン設定や，スプライトの表示などのファンクションがひととおり揃っていて，割と簡単に制御することができるようになっています。たとえば，スクリーンに文字列を表示したいのであれば，BIOSコールのtext_put_stringを使います。C言語でプログラミングを書いているならばこうです。

```
text_put_string(0,0,"Press Any Key to Exit!");
```

ところで，WonderWitchのマニュアルにはFreya BIOSのコール一覧が掲載されているのだが，これらは基本的にすべてアセンブラでの呼び出し方法が記載されています。C言語でプログラムを作る場合には，このアセンブラでの呼び出しに相当する関数の使い方を調べなくてはなりません。実際にはDOS上のgrepなどのツールを使って「*.h」という名前のCのヘッダファイルを調べることになるでしょう(とはいえ，これも規則的ですので，慣れてくればそう難しい話ではないのですが。たとえばfooをいうBIOS APIの引数がaxであれば，Cではfoo (int ax)になります)。

また，BIOS以外にもWonderWitchにはIL (Indirect Library)というライブラリ群があります。これは，各プログラムから共通に呼び出すことのできる動的なライブラリで，標準でILライブラリがいくつかCD-ROMに収められていますし，もちろんユーザーがILを作ることもできるようになっています。なお，このこの標準ILを使ったプログラムを使用する場合，ユーザーはアプリケーションとILの両方をWonderSwanに転送しておく必要があります(ただし，WonderSwanのファイルシステム上にあれば，アプリケーションは勝手にILを探し出して使用してくれるので，アプリケーション側から特に呼び出しなどが必要になることはないですが)。

標準で提供されるライブラリなどに関しては，BIOSは低級な機能を提供しILが高級な機能を提供する，という役割になっているようで，たとえば，サウンド機能でいえば，BIOSコールではサウンドレジスタを直接に操作するような関数が揃えられており，SoundILではMMLデータをセットすることで音楽が演奏できるというふうになっています。

ちなみに，このFreyaOSには「Meg」というユーザーインタフェイスが添付されていて，ユーザープログラムの転送などWonderSwan側からの操作はこのMegで行うのですが，実はこのMegもFreyaOSのILのひとつという形を取っています。ですので，ユーザーはILの作成方法さえ知っていれば，Meg代替のシェルを作るのも，そう難しいことではありません。

図6

図7

実際のプログラミング

さて，それでは，実際にWonderSwan用のごく，簡単なプログラムを作ってみしょう。今回作るのは，「スカッシュ」。要するに一人テニスです。

画面の中に壁と，フィールドがあります。その中にボールが飛んでおり，これをラケットに当てるわけです。壁とフィールドはバックグラウンドにキャラクタを配置します。ボールは16*16ドットつまり4個分のスプライト。ラケットも8*32で4個分のスプライトです。

ラケットにボールがあたると「ブッ」という音がします。ラケットはXボタンの上下で動きます。

プログラムの終了は「START」ボタンです。

・グラフィック周りのプログラムとデータを作る。

WonderWitchには，bmpconvという名前のWindowsビットマップファイルからのコンバータがあります。このプログラムは，8×8ドット単位の大きさのWindows 256色ビットマップファイル(ただし，使用色数は4色であること)を白黒灰色透過色，もしくは白黒4階調のキャラクタデータに相当する16進データを作ってくれますので，このデータをプログラムリスト本体やヘッダファイル中に埋め込んだりして使うと便利でしょう。あるいは，リソースファイルとして持つこともできます。

たとえば，ソースコード中に直接ハードコートするためのリストを出力させてみましょう。そのために，「format」というファイルが必要になります。これはどのようにコンバートしたファイルを残しておくかを決めるファイルです。以下のように書いておきます。

```
#define %s_width %d
```

```
#define %s_height %d

unsigned %s bmp_%s[] = {
```

そして,

```
bmpcnv -c ファイル名.bmp
```

として，このツールを起動すると,

```
ファイル名.h
```

として以下のようなファイルができます。

```
#define ball16_width 2
#define ball16_height 2
static unsigned short bmp_ball16[] = {
0x0007, 0x001F, 0x003F, 0x007D, 0x007F, 0x00F7,
0x00F7, 0x00FF,
0x00E0, 0x08F0, 0x18E4, 0x04FA, 0x0CF2, 0x03FC,
0x03FC, 0x03FC,
0x00FF, 0x00FF, 0x00FF, 0x007F, 0x601F, 0x3847,
0x0738, 0x0708,
0x03FC, 0x03FC, 0x07F8, 0x1CE2, 0x1CE2, 0x7C82,
0xF804, 0xE000,
};
```

ちなみに，これは16×16ドットの大きさのBMPファイルが，4階調カラーのキャラクタ2×2個分のデータに変換されたデータなのですが，このリストをアプリケーションの本体プログラムにコピーするか，本体プログラム中で,

```
#include "ファイル名.h"
```

として,

```
font_set_colordata(128, 1, bmp_checker);
```

とするとキャラクタが作成できます。

なお，sprite_set_charでは，スプライトにキャラクタのデータをコピーすると同時に，そのスプライトの設定も行います。ですので，使い方としては,

```
sprite_set_char(スプライト番号,属性データ | スプライト番号);
```

と考えるといいでしょう。属性部分はこのように計算します。

```
(Hm * 0x8000)|(Vm * 0x4000)|(Pr * 0x2000)|(Ct *
0x1000)|((Palette-8) * 0x200)
Hm ... 横反転なら1,しないなら0
Vm ... 縦反転なら1,しないなら0
Pr ... 1でスクリーン2より上にスプライトが表示。そうでないなら
0
Ct ... 1でスプライトウィンドウの外側部分を表示。そうでないな
ら0
Palette ... 使用するパレット番号(8-11はそのままのカラーパレ
ットとして使われる。黒の部分を透明として扱うならは12-15を
使用すること)。
```

サンプルプログラムでは，あらかじめ属性部分を計算しておいて,

```
sprite_set_char(i * 4, 0x0800 | 129);
```

のように使っています。

で，スクリーンですが,

```
screen_fill_char();
```

などで画面に配置できます。

スプライト画面に表示するには,

```
sprite_set_char(0,128);
```

でキャラクタ番号128の定義内容をスプライト0番に定義し,

```
sprite_set_range(0,1);
```

でそのスプライトの画面への表示を許可して,

```
sprite_set_location(0,0,0);
```

とすることでで座標(0,0)にスプライト0番として表示されることができます。

このスプライトを移動させたいときは，たとえば，先に座標(0,0)に表示されていたスプライト0番をほかの座標(52,40)に表示したいときは，再びsprite_set_location()関数を使って,

```
sprite_set_location(0,52,40);
```

とするだけでOKです。

・サウンド機能を使う

サウンド機能を，直接サウンドレジスタを使用して使う場合は，まず，16レベル(4ビット)×32ステップPCM波形データを，16バイトのデータでチャンネルに登録します。それからピッチ，ボリュームを設定することでその音色の音を出すことができます。

まず，波形データを作るのですが，これはWonderWitchには特にツールなどはツールは付属していません。ですので，手作業でこのデータを作ることになります。といっても，構造は簡単です。これは要するにある時間を32分割した単位で，それぞれの時間でのボリュームを4ビットで表して，2単位ごとに1バイトで表しています。

もしわからない場合は，ちなみにマニュアルにも記述されていますが，とにかく，サイン波,

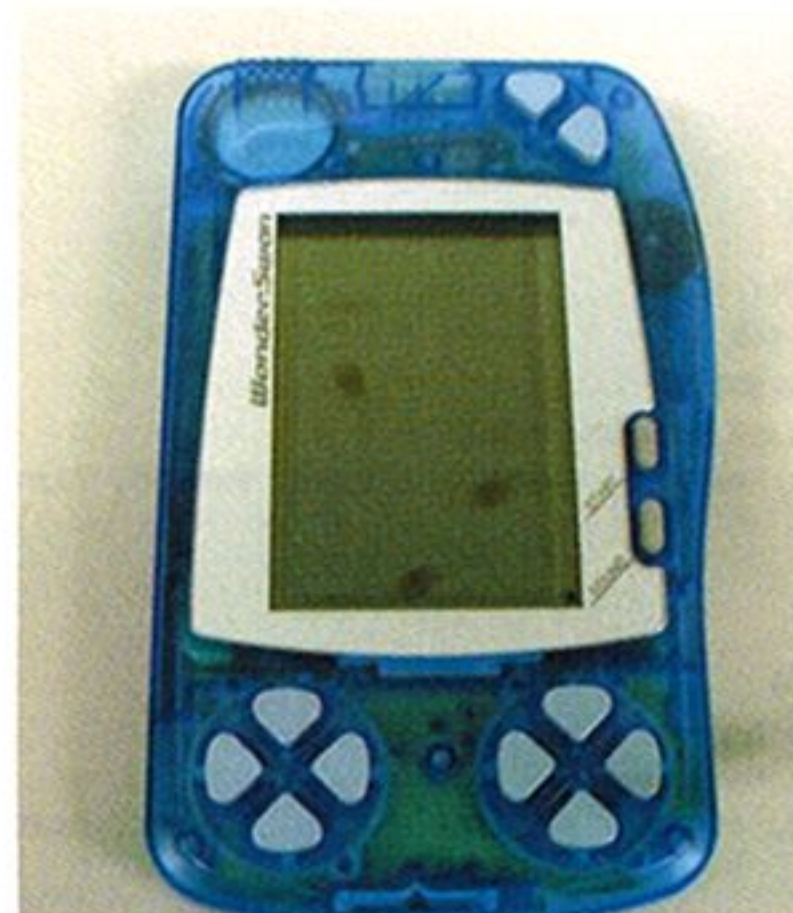



図8　完成したスカッシュゲーム

```
/＊サイン波パターン＊/
static unsigned char wav[]={
0xa8,0xdc,0xee,0xde,0xbc,0x9a,0x89,
0x67,0x56,0x34,0x12,0x01,0x11,0x32,0x75
};
```

あるいは，ダダダダダっという音（矩形パターン）で，

```
/＊矩形波パターン＊/
static unsigned char wav[]={
0xff,0xff,0xff,0xff,0xff,0xff,0xff,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

などで登録してしまうのが簡単でしょう。

そして，プログラムの最初の部分（初期化ルーチンなどに入れておくのが一般的でしょうが）あとは，サウンド周りの初期化（サウンドBIOSの初期化，チャネルのモード設定）などを行ったあと，このデータを，

```
sound_set_wave(0,wav); (チャネル0と波形データを結び付ける)
```

で使いたい音声チャネルを付けておいてください。そして，プログラム中で実際に音を出す場面で，そのチャネルのピッチとボリュームをセットすれば音が出ます。

```
sound_set_pitch(0, 430);
sound_set_volume(0, ff);
```

これはピッチ，ボリュームを0にするまで鳴り続けますので，サウンドを消したいときには明示的に0をセットしなければなりません。

このサンプルプログラムでは，カウンタを作って，音が鳴り始めてからメインループ中で決まった回数繰り返したらその音を止めるようにしています。

・ボタンを検知する

ボタンは「キーBIOS」を操作することによって見ることができます。ちなみにWonderWitchではBIOSで処理されたボタンの情報のことをキーと呼んでいます。ですので，ボタンを見る，といえばキーを操作することとはほぼイコールです。キーの内容は，

```
key = key_press_check();
```

などとして知ることができます。この変数keyは各ビットがボタンの押されているかいないかを示しています。マニュアルのキーBIOSの解説に何ビット目がどのボタンを示すかが表示されています。

ただ，Cコンパイラでプログラミングをする場合にはヘッダファイル中で各ボタンがラベルとして定義されているのでそちらを使うほうが読みやすいリストにすることができるでしょう。内容はkey.hで定義されていて，それぞれ以下のようになっています。

```
KEY_START　スタートボタン
KEY_A Aボタン
KEY_B Bボタン
KEY_UP1 Xボタンの上
KEY_RIGHT1 Xボタンの右
KEY_DOWN1 Xボタンの下
KEY_LEFT1 Xボタンの左
KEY_UP2 Yボタンの上
KEY_RIGHT2 Yボタンの右
KEY_DOWN2 Yボタンの下
KEY_LEFT2 Yボタンの左
```

つまり，Xボタンの上が押されている場合の処理はこのようになります。

```
if( KEY_UP1 == (key & KEY_UP1)){
  上の場合の処理

}
```

ということで……

ということでできたのが，リスト1のサンプルプログラムの（samplesq.c）です。リスト2は設定ファイルということになります。これをコンパイルし

リスト1

```
---------- samplesq.c
[1]#include <sys/bios.h>
[2]#include <stdlib.h>
[3]
[4]/*--------------------------------------------*/
[5]/* Sample Program on WonderSwan               */
[6]/*              for Oh!X                      */
[7]/*   By (de). 2000                            */
[8]/*                                            */
[9]/* 使用、改変、配布…御好きなようにどうぞ。     */
[10]/*--------------------------------------------*/
[11]
[12]
[13]/*グローバル変数*/
[14]int x[3],y[3];
[15]int dx[3];
[16]int dy[3];
[17]int bx,by;
[18]
[19]/*格子模様のパターン*/
[20]#define checker_width 1
[21]#define checker_height 1
[22]static unsigned short bmp_checker[] = {
[23]    0xFF00, 0xFF00, 0xFF00, 0xFF00, 0xFF00, 0xFF00, 0xFF00, 0xFF00,
[24]};
[25]
[26]/*バーのパターン*/
[27]#define bar_width 1
[28]#define bar_height 1
[29]unsigned short bmp_bar[] = {
[30]    0xFEFE, 0xFEFE, 0xFEFE, 0xFEFE, 0xFEFE, 0xFEFE, 0xFEFE, 0xFEFE,
[31]};
[32]
[33]/*ボールのパターン*/
[34]#define ball16_width 2
[35]#define ball16_height 2
[36]
[37]static unsigned short bmp_ball16[] = {
[38]    0x0700, 0x1F00, 0x3F00, 0x7D00, 0x7F00, 0xF700, 0xF700, 0xFF00,
[39]    0xE000, 0xF800, 0xF804, 0xFC02, 0xFC02, 0xFF00, 0xFF00, 0xFF00,
[40]    0xFF00, 0xFF00, 0xFF00, 0x7F00, 0x7F00, 0x3F40, 0x0738, 0x0708,
[41]    0xFF00, 0xFF00, 0xFF00, 0xFC02, 0xFC02, 0xFC02, 0xF804, 0xE000,
[42]};
[43]
[44]/* サイン波データ … MagicalBookより転載 */
[45]static unsigned char wav[]={
[46]    /*0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00*/
[47]    0xa8,0xdc,0xee,0xde,0xbc,0x9a,0x89,
[48]    0x67,0x56,0x34,0x12,0x01,0x11,0x32,0x75
[49]};
[50]
[51]
[52]/*ボールの表示*/
[53]void display_ball(int i)
[54]{
[55]    sprite_set_location(i*4,  x[i]+0,y[i]+0);
[56]    sprite_set_location(i*4+1,x[i]+8,y[i]+0);
[57]    sprite_set_location(i*4+2,x[i]+0,y[i]+8);
[58]    sprite_set_location(i*4+3,x[i]+8,y[i]+8);
[59]
[60]}
[61]
[62]/*バーの表示*/
[63]void display_bar(void )
[64]{
[65]    int i;
[66]
[67]    for(i=0;i<4;i++){
```

```
[68]        sprite_set_location(12+i,bx,by+(i*8));
[69]    }
[70]}
[71]
[72]
[73]/*グローバル変数などの初期化*/
[74]void init_val()
[75]{
[76]    int i;
[77]
[78]    /*乱数系列の初期化*/
[79]    srand(sys_get_tick_count());
[80]
[81]    for(i=0;i<3;i++){
[82]        x[i] = 150;
[83]        y[i] = rand() % 128;
[84]        do{
[85]            do{
[86]                dx[i] = rand()%3 *(-1);
[87]            }while(dx[i]==0);
[88]            do{
[89]                dy[i] = rand()%5 - 2;
[90]            }while(dy[i]==0);
[91]        }while((dx[i]==0)&&(dy[i]==0));
[92]    }
[93]
[94]    /*bx=208;*/
[95]    bx=170;
[96]    by=60;
[97]}
[98]
[99]/*サウンドを鳴らすルーチン*/
[100]void sound_on(int x){
[101]    sound_set_pitch(0, 440*x);/*周波数セット*/
[102]    sound_set_volume(0, 0xff);/*ボリュームセット*/
[103]}
[104]
[105]/*サウンドを消すルーチン*/
[106]void sound_off(){
[107]    sound_set_volume(0, 0);/*消音*/
[108]    sound_set_pitch(0, 0);
[109]}
[110]
[111]
[112]/*メイン関数*/
[113]void main(){
[114]
[115]    int i,k;
[116]    int key;
[117]
[118]    init_val();
[119]    k=0;
[120]
[121]    /*---- サウンド初期化 ----*/
[122]    sound_init();
[123]    sound_set_wave(0,wav);
[124]    sound_set_output(0x0f);
[125]    sound_set_channel(0x03);
[126]
[127]    /*---- 画面初期化 ----*/
[128]    text_screen_init();
[129]
[130]    /*ディスプレイコントロール*/
[131]    display_control(0x0F05);
[132]
[133]    /*LCDカラーの設定*/
[134]    lcd_set_color(0x0444,0xffff);
[135]
[136]    /*---- キャラクタ関係 */
[137]    /*キャラクタパレット番号*/
[138]    /*(注：パレット0-7はキャラクタ専用、8-15はスプライトと共用)*/
[139]    /*パレット12～15を使うと0x00は透明になる*/
[140]    /*キャラクタに透明部分を作る場合はパレット12～15を使うこと*/
[141]    palette_set_color(12,0x048c);
[142]
[143]    /*キャラクタを定義*/
[144]    font_set_colordata(128, 1, bmp_checker);
[145]    font_set_colordata(129, 4, bmp_ball16);
[146]    font_set_colordata(133, 1, bmp_bar);
[147]
[148]    /*---- スプライト関係 */
[149]    /*スプライトの表示範囲を設定*/
[150]    sprite_set_range(0,16);
[151]
[152]    for(i=0;i<3;i++){
[153]        /*スプライトをキャラクタの内容で定義*/
[154]        /*「0x0800」はパレット12番に対応*/
[155]        sprite_set_char(i*4,  0x0800 | 129);
[156]        sprite_set_char(i*4+1,0x0800 | 130);
[157]        sprite_set_char(i*4+2,0x0800 | 131);
[158]        sprite_set_char(i*4+3,0x0800 | 132);
[159]        /*スプライトで定義したボールを表示*/
[160]        display_ball(i);
[161]    }
[162]
[163]    /*バーをキャラクタ内容で定義*/
[164]    for(i=0;i<4;i++){
[165]        sprite_set_char(12+i,0x0800 | 133);
[166]    }
[167]    /*スプライトで定義したボールを表示*/
[168]    display_bar();
[169]
[170]
[171]    /*---------------- 画面表示 -----------------*/
[172]
[173]    /*背景を表示*/
[174]    screen_fill_char(0,1,1,26,16,0x0800 | 128);
[175]
[176]    do{
[177]        int j;
[178]
[179]        key = key_press_check();
[180]        if( key == KEY_START) break;
[181]        if( KEY_UP1 == (key & KEY_UP1)){
[182]            by = by-1;
[183]            if(by<0){ by=0; }
[184]        }else if( KEY_DOWN1 == (key & KEY_DOWN1)){
[185]            by = by+1;
[186]            if(by>102){ by=102; }
[187]        }
[188]        display_bar();
[189]
[190]        if(k>0){
[191]            k=k-1;
[192]            if(k==0){
[193]                sound_off();
[194]            }
[195]        }
[196]
[197]        for(j=0;j<3;j++){
[198]            x[j]=x[j]+dx[j];
[199]            y[j]=y[j]+dy[j];
[200]
[201]            if(x[j]<0){  x[j]=0;  dx[j]=dx[j]*(-1);}
[202]            if(x[j]>208){x[j]=208;dx[j]=dx[j]*(-1);}
[203]            if(y[j]<0){  y[j]=0;  dy[j]=dy[j]*(-1);}
[204]            if(y[j]>128){y[j]=128;dy[j]=dy[j]*(-1);}
[205]
[206]            /*衝突判定*/
[207]            if(((x[j]+16) > bx)&&(x[j] < (bx+8))){
[208]                if(((y[j]+8) > by) && (y[j]<(by+32))){
[209]                    /*あたった*/
[210]                    dx[j]=dx[j]*(-1);
[211]                    dy[j]=dy[j]*(-1);
[212]                    k=3;
[213]                    sound_on(10+j);
[214]                }
[215]            }
[216]            /*スプライトで定義したボールを表示*/
[217]            display_ball(j);
[218]        }
[219]
[220]        /*ボールをゆっくり見たいときは↓をコメントアウトしてください。*/
[221]         sys_wait(1);
[222]    }while(TRUE);
[223]
[224]}
```

てできるのがWonderWitch用バイナリsamplesq.fx（スカッシュゲーム）です（付録CD-ROM中には，このsample.fxも含まれています）。このsample.fxをWonderWitchに転送するとプレイすることができるようになるわけですね。

さて，これでプログラムが作れるようになるまでのお話はおしまいです。プログラミングの方法さえわかってしまえば，あとは使い方次第でいろいろな面白いことができるはずです。

まだまだ，WonderSwan，WonderWitch，FreyaBIOSにはテキスト表示，拡張ポート（ワンダーゲートを使っての赤外線の送受信などもできます）など使うことのできる機能がたくさんあります。皆さんも，ぜひ，このWonderWitcプログラミングを楽しんでみてください。

リスト2

```
---------- samplesq.cf
[1]name: samplesq
[2]info: スカッシュ(サンプル)
[3]mode: 7
[4]source: samplesq.bin
[5]output: samplesq.fx
```

Palmプログラムを作る（第1回）

石上　達也　Ishigami Tatsuya

世界的に見て，現在の携帯端末でトップを走るのがPalmプラットフォームによるものです。ここではPalmプログラミングを始めるに当たっての基礎知識とメーカーに用意された基本的な開発環境を見ていきましょう。

「充実した環境下にあってもなお，Palmデバイス用のアプリケーションを作成することは，大きなチャレンジです。Palmデバイスにはわずかなメモリしかなく，画面も小さく，文字を入力する手段も限られています。……(中略)……このような状況で動作するアプリケーションの作成は，ユーザーインターフェースの設計，実現機能の決定，そしてコーディングのどれをとっても開発者に多くを要求します。場合によっては，デスクトップPC上での開発で得た多くの常識と知識を捨てる必要があるかもしれません。
　しかし，その大きなチャレンジを成し遂げたとき，PC用のアプリケーションとは次元が異なる喜びをそのユーザーと開発者は得ることができるでしょう」　　　　　　（山田達司「『Palm プログラミング』発刊によせて」より）

PDAに関しては私はWiz派なので，Palmの携帯性は特に問題ではありませが，少ないメモリ，非力なCPUとプログラム開発の箱庭としては大変魅力的です。WindowsのAPIが数千に及び，アプリケーションの個性が「どれだけアルゴリズムを知っているか/見つけたか」ではなく，「どれだけMS製のAPI/クラスを知っているか」や「どれだけ熱心にMSDN（Microsoft Developers News）を読んでいるか」に移りつつあるいま，久しぶりに面白くプログラミングができる環境です。

技術的には，SX-Windowとほぼ同程度，環境はCode Warriors（後述）と申し分ありません。また，Windows3.1の頃のようにセグメントメモリモデル云々など理不尽なこともありません。また，ユーザーの数は世界中に数百万人と「Oh!X」の最盛期の10倍以上います。「箱庭」の大きさにも問題ありません。

「携帯機」という名のもとに，箱庭が帰ってきました。

Palmとはなにか

実は，（U）氏から「こんなもの出たらしいんだけど，入手できないかなぁ」といって，Visorの入手を頼まれたのが，私の本格的なPalmとの出会いでした（といっても，互換機ですが）。旧「Oh!X」休刊号の座談会でも，こそっと漏らしていたように，当時から携帯機に注目はしていたのですが本命はHP200LXかLibretto（＋軽い独自OS，IBMのWeb Boyはよいセンいっていたと思ったのだが……）と考えていたので，Palm関連はあまり注意を払っていませんでした。そんなわけで，あまり詳しくはないのですが，以下に簡単にまとめてみます。

まず，Palm Computingという会社がありました。この会社は「Graffiti」という独自のペン入力モジュールを作っていました。どのくらい独自かというと，「A」と入力するのに「∧」と入力する必要があったり，「K」を「〆」で入力するというようにとても独自なのですが，ほとんどの文字が一筆書きでき，抽出すべき特徴点も3カ所以下というコンピュータにとって処理しやすいものでした。これは，Apple社のNewtonに採用されたようです（この頃，漢字認識を軽々とやってたシャープは偉大ですね。テンコ盛りのザウルスだけでなく，もっとWizにも力を入れてくれればよかったのに……）。その後，Palm Computingは，Graffitiの外販だけでなく，PDAそのものを発表します。

Palm Computingは，その後，U.S.Roboticsに買収されて，U.S.Roboticsが3Comに買収されて，Jeff Hawkins氏をはじめとするPalmの主力開発者が，独立してHandspringを設立して，やっぱり3ComもPalm部門をスピンアウトさせて，などなどドラマがあるらしいのですが，個人的に思い入れがないため，詳細は省略して結果をまとめると，だいたい以下のようなマシンが発売されてきました。

年を追うごとに，メモリが増加したり，IrDAがサポートされたりしていますが，基本的にLCDの160×160ドットという解像度など，基本的な仕様は一定ですし，アプリケーションプログラムの下位互換性はほぼ保証されています。

Palm OSは，3Com社以外にもライセンスされており，Nokiaの携帯電話や，Symbolのバーコードリーダなどに統合されたものや，IBMのWork PadのようなOEM品，HandspringのVisorのような互換機など，上記以外のいろいろなマシンに採用されています。現在ではソニーもライセンスを取得し，メモリースティックを使った商品を発売していますね。

GoのPen PointやGeneral MagicのMagic Cap以来，PDAといえば「なぜかデスクトップ画面に机の絵があって，左脇にメモ帳やアドレス帳が」という画面で完結したものが主流を占めていました。しかし，Palmでは，データ入力は主役はあくまでPCを想定し，リソースの限られたPalmはそのデータを回覧するための「触手」という割り切ったスタンスで，バランスのよい操作体系を実現しました。また，ペン入力一辺倒でなく，よく使う機能（主にショートカットキー）はボタンに割り付けるなどという点でもユニークです。

CPUには，モトローラの68（EZ）328というMC68000によく似たものが採用されています。詳しくは，囲み記事を参照していただくとして，大雑把にいって，同一クロックのMC68000よりも1.5倍ほど高速なようです。

音楽や画像などは及びませんが，CPUとメモリという基本部分だけに注目すれば，X68030＋SX-WindowでできたことはPalm 上でもできるはずというセンが一応の基準といえるでしょう。

開発に必要なもの

●PCあるいはMacintosh

Palm OS自身はメモリの少ない，消費電力の少ない環境で効率よく動くように設計されているので，大規模なアプリケーションは動きません。また，独自のペン入力システムGraffitiもキーボード入力に比べて使いやすいとはいいがたいですし，画面表示も160×160ドットで大きなファイルを編集するには便利な環境ではありません。

このため，プログラム開発はPalmマシンでなくPCあるいはMacintosh上にて行います。GNUの各種ツールを用いてUNIXを使えないこともないのですが，あまり初心者向けでないため本稿では割愛します。

● Code Warrior for Palm OS

Palmは，モトローラ社のCold Fire（実際には周辺回路をまとめたDragonball）という68000に似たCPUを使用しており，その仕様は広く公

表1　Palmの歴史

名称	発表年	クロック周波数	メモリ	IrDA	OS Version
Pilot 1000	1996	不明	128K	×	1.0
Pilot 5000	1996	不明	512K	×	1.0
Palm Pilot Personal	1997	不明	512K	×	2.0
Palm Pilot Professional	1997	不明	1024K	○	2.0
Palm III	1998		2048K	○	3.0
Palm IIIc	1999	20	8192K	○	3.5
Palm IIIe	1998	16	2048K	○	3.1
Palm IIIx	1998	16	4096K	○	3.1
Palm V	1999	16	2048K	○	3.1
Palm Vx	1999	20	8192K	○	3.3
Palm VII	1999	16	2048K	○	3.2

開されています。ですから，原理的にはCold Fireの開発環境であれば，どんなコンパイラやアセンブラでも使用できるはずです。事実，初期の頃はGNU C Compiler（以下gcc）が標準開発ツールとして使われていました。

ただし，Palm OSの場合はアプリケーションがプログラムコードだけでなくリソースデータとともに成り立っているので，両者を共に扱える統合環境のほうが何倍も効率的です。リソースデータとはウィンドウのどこにどのようなボタンを配置するかなどを収めたデータで，対象が主にGUIオブジェクトですので，

GUI的にマウスで編集できるか

あるいは，

テキストエディタで逐次数値を打ち込みながら開発するか，

では作業効率が数倍違います。

これらの事情はWindowsと同じです。gcc自身がどんなに優れていても，結局，全体の使いやすさからVisual C++へと流れていったのでした（そのような統合環境のなかったSX-Windowの開発者は，RLK/RSCというテキストベースのリソースエディタで，一所懸命にリソースデータを作成していましたね）。

現在，標準開発ツールとして，Palm OS開発元のPalm Computingが想定しているのは，Metrowerks社の「Code Warriors for Palm OS」という統合開発環境です。以前は，その評価版の「Code Warriors Lite for Palm OS」というのがあったのですが最近ではなくなってしまったようです。残念ながら，この記事を参考にされる方は「Code Warriors for Palm OS」の製品版を購入してください。

Macintoshでは，初期の頃，Think CというCコンパイラが標準的に使われていましたが，Power Mac発表あたりからCode Warriors for Macintoshが盛り上がりを見せ，現在では完全にMacintoshの標準開発ツールとなった感があります（詳しくは30ページの記事や本誌復刊1号で柴田氏の紹介記事を参照ください）。そういうわけで68000系のプラットフォームには特に定評があります。

●Palmマシンあるいはエミュレータ

開発環境と実行環境が異なる場合，デバッグの方法は2種類あり，

a）作ったプログラムを実機（Palmマシン）上に転送して実行
b）PC上にPalmマシンをエミュレートする環境を用意し，そこで実行

のどちらかを用います。

a）は，CD-ROMやインターネット上からPalmアプリケーションを拾ってくる場合と同じで，手順も簡単ですが，プログラムの転送に時間がかかります。デバッグ中は何度も作りかけのプログラムを転送するので，この方法はあまり効率的ではありません。また，プログラムにバグがあってもボタンが6つ，液晶が160×160ドットしかない環境では，なにが問題なのか特定が難しいですし，プログラムが暴走してしまうと大事なデータ（アドレス帳など）を破壊してしまうという可能性もあります。

それに対し，b）のエミュレータを使う方法は，

ハードディスクからプログラムを直接読み込むので転送に要する時間がほとんどかからない

Cold Fireそのものではなく，Cold FireのコードをPentiumが解釈しながらプログラムが実行されるので，不正な命令やメモリアクセスは事前に弾ける。また，Palm OS自身のメモリマネージャとは別にエミュレータもメモリ管理情報を記憶しておくので，この手のプログラムにありがちなメモリリークを高い確率で発見できる

プログラム中にBreak Pointを設定し，変数の変化などを見ながらステップ実行できる

後述のPOSEというエミュレータを使用した場合，Gremlinsという自動ストレステストを行えるので潜在的なバグの発見が容易

などのメリットがあります。

逆に，エミュレータを使う場合に考えられるデメリットとして，

実機と比較して実行速度が遅い。

ということが考えられますが，Palm自身がそんなに高速な機械でもありませんし，X68000のエミュレータがPC上では実機よりも速かったという事実を考えてもさほど問題にはならないでしょう。逆に，ゲームなどの場合はエミュレータでは速すぎて実機とは動作が異なってしまうということも，近い将来あるかもしれません。現在，私は300MHzのCelron機で使用していますが，体感で実機の半分のスピードは出ているかな，といったところですので最近のマシンでならすでに実機以上の速度かもしれません。

POSE

Palm用エミュレータにもいろいろ種類があるのですが，本稿ではいちばんメジャーと思われるPOSE（Palm OS Emulator）を使用します。

これは，以前Copilotと呼ばれていたフリーソフトウェア（正確にいうとGPLベースの）をPalm Computingがライセンスしているもののようです。

現在の最新版ビルドは，

http://www.palm.com/devzone/pose/pose.html

にあるはずですので，ダウンロードしてください。*1

Visorのような Palm互換機のユーザーは，ひょっとしたら互換性の問題に引っかかるかもしれないので，製造元Handsprings社のWebページもチェックしてパッチが公開されていないか調べておきましょう。私の場合は，POSE 3.0a3で後述のROM吸い出しができないなぁと思っていたら，Handspring社のweb Pageにパッチが公開されていました。

http://www.handspring.com/developers

なお，

http://www.palm.com/devzone/pose/seed.html

に公開デバッグ中のPOSE最新版がありますが，初心者のうちは手を出さないほうが賢明でしょう。

また，Macintoshユーザーには「Palm Simulator」という同様のプログラムがあって，PalmのCPUと（昔の）MacintoshのCPUは，割と似ているので，高速なエミュレーション（シミュレーション）が可能だそうです。が，使ったことがないので今回は割愛させてください。

*1：以前DOS/V magazineの付録CD-ROMなどで配布されていたCode Warrior Lite for Palm OSをインストールした方は，そのなかにPOSEも含まれていますが，バージョンが古いもののようです。Code WarriorとPOSEはほとんど独立したプログラムですので，バージョンの組み合わせによる相性の問題はほとんどありません。別ディレクトリの（可能ならURL上で最新の）POSEを使用するようにしてください。

Palm OSのイメージ

このPOSEは，Palmマシンのエミューレートを行うだけでOSの機能は含まれていません。Palm OSも含めて，Cold Fireのプログラムをエミュレートします。X68000エミュレータがあってもIOCSイメージやブートディスクがないとWindows上でX68000プログラムが走らないのと同じです。

POSE用にPalm OSのイメージを用意する方法は以下の2つがあります。

a）実際のPalm Pilotから吸い上げる
b）Palm Computingに頼んでROMのイメージをライセンスしてもらう（無料）

a）は，書類にサインする必要もありませんし，その日のうちに準備をすることができるのでお手軽です（図1-1〜1-6を参照してください）。ただし，私が使ったときは（2000/3/20），この吸い出しツールはPCのシリアルポート経由でしか，イメージの転送ができないようで，Visorのように USBグレードルしかPCとリンクする手段がない機種では使えませんでした（あるいは，オプションのシリアルグレードルを購入するか*2）。

b）の方法は，互換機を含めてすべてのPalmマシン/ユーザーが使える方法です。極端な話，実際にPalmマシンを持っていなくてもプログラムを作成できます。

ただ，残念ながらWeb経由で直接ダウンロードすることはできず，以下のようにライセンス云々の書類にサインをして審査を経なければ入手できません。

1）http://www.palm.com/devzonve/へ行き，Palm Pavilionの会員となり，ユーザーIDとパスワードをもらう
2）Palm Pavallionのホームページへ行き，ROM Imageのコーナから "Guidelines for submitting PROTOTYPE LICENSE AND CON

FIDENTIALITY AGREEMENT"という契約書をダウンロードし，内容を吟味のうえ，２通印刷して両方にサインし，米国のPalm Computingか日本のスリーコムジャパン（株）へ郵送する（emailとFAXは不可）

3）だいたい３～４週間くらいで，ダウンロード先を示したURLとユーザーID，パスワードが送られてくるので，そこからダウンロードする

このROM Imageを使うと，

a）APIへの不正なパラメータをチェックすることができる
b）アプリケーション終了時に解放し忘れたメモリハンドルをチェックできる
c）ロック忘れたメモリハンドルへのアクセスに対し，警告が出る
d）宣言されていないメモリエリアへのアクセスをチェックできる

と，なにかと便利です。

特にｂやｃの問題は，プログラム自身は通常に動作するわけですから，デバッグ時に発見することは困難です。長く使っていると，ｂの要因でアプリケーションプログラムが不安定になるとか，ｃの要因で，あるアプリケーションと特定の組み合わせで動作がおかしくなるなど，再現性の低い問題として症状が現れて対応がやっかいです。ところが，POSEを使用すれば，エミュレータ上でこのような問題を引き起こす可能性のあるアクセスをあらかじめ教えてくれるのでとても便利です。

＊2：11月吉日，突然，運送会社の成田通関事務所から電話があった。
「お届けものなんですが，ビザールってなんですか？」
「は？？　誰から着てます？」
「えっと，ニュージャージ州の●×？？さん，からなんですが」
一瞬，なぜ●×？？氏がアメリカ産お下劣雑誌を送りつけてきたのか考える。
「エー・ビー・シーでいってもらえます？」
「えっと，V・I・S・O・Rっと。なんですかこれ？」
「V・I・S・O……あっ，バイザーね。バイザー。電子手帳みたいなもんです」
「あ，手帳ね。手帳。いやね，ちょっと品目によって関税率が変わってくるから。手帳なら，非課税と」
「……（無言）」
「それとねぇ，一緒にシリアルケーブルって書いてあるけど，これなに？」
「あっ，銅線みたいなもんです」
「銅線って，銅の線のこと？」
「はい」
「じゃあ，同軸ケーブルね」
なんで，そんな専門用語を知っているんだと思いつつ，
「ええ，そんなもんです」
「じゃあ，課税品目。物品消費税2600円と。通関時は弊社で立て替えときますんで，都合のいいときに，振り込んでおいてください。請求書，同封しときますんで」
「はっ？ 消費税2600円？」
「だってあなた，さっき同軸ケーブルの一種っていったでしょう」
「いや，似てはいますが，ぜんぜん別物のはずで，そんなに課税されるものでは，ないはずなんですが……」
「いや，こっちはちゃんと計算していますよ」
「いや，どちらかというと，ヘッドホンのコードに近いような，あれなら個人輸入は非課税でしょう？」
（真偽のほどは確かでない。とっさに思いついた）
「でも，同軸ケーブルの一種なら課税対象です」

写真1　Palm OSの吸い上げ方

Palm Desktopにて，使用しているComポート番号，ボーレート，プロトコルを確認する（Hot Sync → Setupメニュー）

Setup
General | Local | Modem | Network
Serial Port: COM1
Speed: 57600
OK　キャンセル　適用(A)　ヘルプ

POSEのパッケージに入っているROM Transfer.prcというプログラムをPalmへ転送する。転送終了後，Palm Desktopを終了する（Comポートを開放するため）

Install Tool
User: Tatsuya Ishigami
File(s) listed below will be installed on your handheld the next time you perform a HotSync operation:
File Name | File Size
ROM Transfer.prc | 5K
Add... Remove... Done
Tips:
Find other applications to install on your handheld at http://www.handspring.com
The 'Add' button looks first in the ¥ADD-ON folder inside your C:¥PROGRAM FILES¥PALM folder. This folder is a convenient place to store downloaded handheld files.

転送した「Transfer ROM」をPalmマシンで起動し，1-1で調べたものと同じ転送スピードを設定。「Begin Transfer」を押す

ROM Transfer
ROM Transfer is provided as a companion utility to the Palm OS Emulator. To use this utility, select "Download ROM..." from the File menu in the emulator and follow the instructions provided.
Transfer Speed: ▼ 57,600 bps
Status: Idle
Begin Transfer

転送中はプログレスバーが表示され，約3～5分かかる。ROM内容転送終了後，ファイル名を尋ねられるので，visor.romなど覚えやすい名前でデータをセーブする

Progress
Transferring...
Cancel

POSEを起動し「Download」を選択

Palm OS Emulator
New　Start a new emulator session
Open　Load a previous emulator session from disk
Download　Download a ROM image from a Palm OS device
Exit　Leave the Palm OS Emulator

1-1で調べたものと同じ値を設定。「Begin」を押す

Transfer ROM
Instructions:
5. Disable HotSync or any other application that may be using the selected serial port.
6. Click the "Begin" button in this dialog. The Palm OS Emulator will wait for your Palm device to respond.
7. Tap the "Begin Transfer" button in the "ROM Transfer" application on the Palm device.
8. The transfer will take several minutes, so please be patient. If an error occurs during the transfer, the application may be unresponsive or may appear to hang. If this happens, wait for a few minutes to give the application a chance to time-out. It should then resume normal operation.
9. After the ROM has been transferred to your desktop computer, you will be asked to save it to a file. Save it with an appropriate name (for example, "Palm III.rom"). Then create a a new emulator session using that ROM.
Begin　Cancel
Port: COM1
Speed: 57,600 bps

「いや，だから……」
（以下，20分間続けたが無駄だった）
教訓：シリアルケーブルを輸入する際，同軸ケーブルではありません，とはっきり通関業者に答えましょう。
（本当に頭にきたときは，意義申立書を提出することも可能だそうです）

Palmプログラムの構造

Palm用のプログラム構造は，ほかのウィンドウシステム（MS-Windows, Macintosh，SX-Window）とよく似ています。画面やキーボード（ペン）入力の基本部分はOSが管理していて，その情報が「イベント」としてアプリケーションプログラムに送られてきます。いわゆるイベントドリブンと呼ばれている方法ですね。

リスト1は，Code Warriorで，新規プロジェクトを作成すると自動生成されるスケルトンプログラムです。スケルトン（骨格）プログラムとは，アプリケーションにとって最低限必要なデータ，関数のみを含んだプログラムで，ユーザーはこれにユーザー関数を肉付けすることにより，アプリケーションを完成させることができます。

リスト1をよく見ると，プログラムに必要な変数などを初期化し，AppEventLoop（void）関数で，システムから送られてくるイベントを監視し，対応する処理ルーチンへ割り振ります（実際にはシステムイベント以外は，AppHandleEvent（&event））関数へ丸投げされています）。

いまは，なにも行わないプログラムですので，監視するイベントの数も少ないですが，やがて，ペンが押されたら～，文字が入力されたら～，というイベントごとの振り分けをこの関数に埋め込んで，対応する動作を実現します。

起動コード

それでは，スケルトンプログラムを順に見ていきましょう。

Palm OSのプログラムでは，PilotMain()関数を初めに実行するという決まりになっています。[*3]

このPilotMain()関数の引数Word cmd, Ptr cmdPBP, Word launchFlagsですが，ソース中のコメントを読むと，

Word	cmd;	launchCode
Ptr	cmdPBP	launchCodeと共に使用されるポインタ
Word	launchFlags	launchCodeの補助情報

ということでcmdが起動コードを持っていて，ほかの2つの引数は起動コードによって，その意味あいが変わるとあります。

起動コードの一部を表2にまとめておきました。

そもそも，起動コードとはなんでしょう。

Palm OSではプログラムを起動する方法が何種類か存在します。もっともオーソドックスな方法はデスクトップからアプリケーションアイコンをタップして起動する方法です。これ以外にも，Date Bookのように，ある決められた時間に起動するように設定されたアプリケーションもあります。

リスト1　Starter.c

```
/***********************************************************************
 *
 Copyright ｳ 1995 - 1998, 3Com Corporation or its subsidiaries ("3Com").
 All rights reserved.

 This software may be copied and used solely for developing products for
 the Palm Computing platform and for archival and backup purposes.  Except
 for the foregoing, no part of this software may be reproduced or transmitted
 in any form or by any means or used to make any derivative work (such as
 translation, transformation or adaptation) without express written consent
 from 3Com.

 3Com reserves the right to revise this software and to make changes in content
  from time to time without obligation on the part of 3Com to provide
notification
 of such revision or changes.
 3COM MAKES NO REPRESENTATIONS OR WARRANTIES THAT THE SOFTWARE IS FREE OF ERRORS
 OR THAT THE SOFTWARE IS SUITABLE FOR YOUR USE.  THE SOFTWARE IS PROVIDED ON AN
 "AS IS" BASIS.  3COM MAKES NO WARRANTIES, TERMS OR CONDITIONS,  EXPRESS OR
IMPLIED,
 EITHER IN FACT OR BY OPERATION OF LAW, STATUTORY OR OTHERWISE, INCLUDING
WARRANTIES,
 TERMS, OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
 SATISFACTORY QUALITY.

 TO THE FULL EXTENT ALLOWED BY LAW, 3COM ALSO EXCLUDES FOR ITSELF AND ITS
SUPPLIERS
 ANY LIABILITY, WHETHER BASED IN CONTRACT OR TORT (INCLUDING NEGLIGENCE), FOR
 DIRECT, INCIDENTAL, CONSEQUENTIAL, INDIRECT, SPECIAL, OR PUNITIVE DAMAGES OF
 ANY KIND, OR FOR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF
INFORMATION
 OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THIS
SOFTWARE,
 EVEN IF 3COM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

 3Com, HotSync, Palm Computing, and Graffiti are registered trademarks, and
 Palm III and Palm OS are trademarks of 3Com Corporation or its subsidiaries.

 IF THIS SOFTWARE IS PROVIDED ON A COMPACT DISK, THE OTHER SOFTWARE AND
 DOCUMENTATION ON THE COMPACT DISK ARE SUBJECT TO THE LICENSE AGREEMENT
 ACCOMPANYING THE COMPACT DISK.

 ***********************************************************************
 *
 * PROJECT:  Pilot
 * FILE:     Starter.c
 * AUTHOR:   Roger Flores: May 20, 1997
 *
 * DECLARER: Starter
 *
 * DESCRIPTION:
 *
 *
 **********************************************************************/
#include <Pilot.h>
#include <SysEvtMgr.h>
#include "StarterRsc.h"


/***********************************************************************
 *
 *   Entry Points
 *
 ***********************************************************************/


/***********************************************************************
 *
 *   Internal Structures
 *
 ***********************************************************************/
typedef struct
  {
  Byte replaceme;
  } StarterPreferenceType;

typedef struct
  {
  Byte replaceme;
  } StarterAppInfoType;

typedef StarterAppInfoType* StarterAppInfoPtr;


/***********************************************************************
 *
 *   Global variables
 *
 ***********************************************************************/
//static Boolean HideSecretRecords;


/***********************************************************************
 *
 *   Internal Constants
 *
 ***********************************************************************/
#define appFileCreator                  'strt'
#define appVersionNum              0x01
#define appPrefID                  0x00
#define appPrefVersionNum          0x01


// Define the minimum OS version we support
#define ourMinVersion sysMakeROMVersion(2,0,0,sysROMStageRelease,0)


/***********************************************************************
 *
 *   Internal Functions
 *
```

```
*****************************************************************/

/*****************************************************************
 *
 * FUNCTION:    RomVersionCompatible
 *
 * DESCRIPTION: This routine checks that a ROM version is meet your
 *              minimum requirement.
 *
 * PARAMETERS:  requiredVersion - minimum rom version required
 *                                (see sysFtrNumROMVersion in SystemMgr.h
 *                                for format)
 *              launchFlags     - flags that indicate if the application
 *                                UI is initialized.
 *
 * RETURNED:    error code or zero if rom is compatible
 *
 * REVISION HISTORY:
 *
 *****************************************************************/
static Err RomVersionCompatible(DWord requiredVersion, Word launchFlags)
{
   DWord romVersion;

   // See if we're on in minimum required version of the ROM or later.
   FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
   if (romVersion < requiredVersion)
      {
      if ((launchFlags & (sysAppLaunchFlagNewGlobals | sysAppLaunchFlagUIApp))
==
         (sysAppLaunchFlagNewGlobals | sysAppLaunchFlagUIApp))
         {
         FrmAlert (RomIncompatibleAlert);

         // Pilot 1.0 will continuously relaunch this app unless we switch to
         // another safe one.
         if (romVersion < sysMakeROMVersion(2,0,0,sysROMStageRelease,0))
            AppLaunchWithCommand(sysFileCDefaultApp,
sysAppLaunchCmdNormalLaunch, NULL);
         }

      return (sysErrRomIncompatible);
      }

   return (0);
}


/*****************************************************************
 *
 * FUNCTION:    GetObjectPtr
 *
 * DESCRIPTION: This routine returns a pointer to an object in the current
 *              form.
 *
 * PARAMETERS:  formId - id of the form to display
 *
 * RETURNED:    VoidPtr
 *
 * REVISION HISTORY:
 *
 *
 *****************************************************************/
static VoidPtr GetObjectPtr(Word objectID)
   {
   FormPtr frmP;


   frmP = FrmGetActiveForm();
   return (FrmGetObjectPtr(frmP, FrmGetObjectIndex(frmP, objectID)));
}


/*****************************************************************
 *
 * FUNCTION:    MainFormInit
 *
 * DESCRIPTION: This routine initializes the MainForm form.
 *
 * PARAMETERS:  frm - pointer to the MainForm form.
 *
 * RETURNED:    nothing
 *
 * REVISION HISTORY:
 *
 *
 *****************************************************************/
static void MainFormInit(FormPtr frmP)
{
}


/*****************************************************************
 *
 * FUNCTION:    MainFormDoCommand
 *
 * DESCRIPTION: This routine performs the menu command specified.
 *
 * PARAMETERS:  command  - menu item id
 *
 * RETURNED:    nothing
 *
 * REVISION HISTORY:
 *
 *
 *****************************************************************/
static Boolean MainFormDoCommand(Word command)
{
   Boolean handled = false;

   switch (command)
      {
      case MainOptionsAboutStarterApp:
         MenuEraseStatus (0);
         AbtShowAbout (appFileCreator);
         handled = true;
         break;

      }
   return handled;
}


/*****************************************************************
 *
 * FUNCTION:    MainFormHandleEvent
 *
 * DESCRIPTION: This routine is the event handler for the
 *              "MainForm" of this application.
 *
 * PARAMETERS:  eventP  - a pointer to an EventType structure
 *
 * RETURNED:    true if the event has handle and should not be passed
 *              to a higher level handler.
 *
 * REVISION HISTORY:
 *
 *
 *****************************************************************/
static Boolean MainFormHandleEvent(EventPtr eventP)
{
    Boolean handled = false;
    FormPtr frmP;


   switch (eventP->eType)
      {
      case menuEvent:
         return MainFormDoCommand(eventP->data.menu.itemID);

      case frmOpenEvent:
         frmP = FrmGetActiveForm();
         MainFormInit( frmP);
         FrmDrawForm ( frmP);
         handled = true;
         break;

      default:
         break;

      }

   return handled;
}


/*****************************************************************
 *
 * FUNCTION:    AppHandleEvent
 *
 * DESCRIPTION: This routine loads form resources and set the event
 *              handler for the form loaded.
 *
 * PARAMETERS:  event  - a pointer to an EventType structure
 *
 * RETURNED:    true if the event has handle and should not be passed
 *              to a higher level handler.
 *
 * REVISION HISTORY:
 *
 *
 *****************************************************************/
static Boolean AppHandleEvent( EventPtr eventP)
{
   Word formId;
   FormPtr frmP;
```

```
  if (eventP->eType == frmLoadEvent)
      {
      // Load the form resource.
      formId = eventP->data.frmLoad.formID;
      frmP = FrmInitForm(formId);
      FrmSetActiveForm(frmP);

      // Set the event handler for the form.  The handler of the currently
      // active form is called by FrmHandleEvent each time is receives an
      // event.
      switch (formId)
          {
          case MainForm:
              FrmSetEventHandler(frmP, MainFormHandleEvent);
              break;

          default:
//              ErrFatalDisplay("Invalid Form Load Event");
              break;

          }
      return true;
      }

  return false;
}


/***********************************************************************
 *
 * FUNCTION:    AppEventLoop
 *
 * DESCRIPTION: This routine is the event loop for the application.
 *
 * PARAMETERS:  nothing
 *
 * RETURNED:    nothing
 *
 * REVISION HISTORY:
 *
 *
 ***********************************************************************/
static void AppEventLoop(void)
{
  Word error;
  EventType event;


  do {
      EvtGetEvent(&event, evtWaitForever);


      if (! SysHandleEvent(&event))
          if (! MenuHandleEvent(0, &event, &error))
              if (! AppHandleEvent(&event))
                  FrmDispatchEvent(&event);

  } while (event.eType != appStopEvent);
}


/***********************************************************************
 *
 * FUNCTION:     AppStart
 *
 * DESCRIPTION:  Get the current application's preferences.
 *
 * PARAMETERS:   nothing
 *
 * RETURNED:     Err value 0 if nothing went wrong
 *
 * REVISION HISTORY:
 *
 *
 ***********************************************************************/
static Err AppStart(void)
{
   StarterPreferenceType prefs;
   Word prefsSize;


  // Read the saved preferences / saved-state information.
  prefsSize = sizeof(StarterPreferenceType);
  if (PrefGetAppPreferences(appFileCreator, appPrefID, &prefs, &prefsSize, true)
!=
      noPreferenceFound)
      {
      }

  return 0;
}


/***********************************************************************
 *
 * FUNCTION:    AppStop
 *
 * DESCRIPTION: Save the current state of the application.
 *
 * PARAMETERS:  nothing
 *
 * RETURNED:    nothing
 *
 * REVISION HISTORY:
 *
 *
 ***********************************************************************/
static void AppStop(void)
{
  StarterPreferenceType prefs;


  // Write the saved preferences / saved-state information.  This data
  // will be backed up during a HotSync.
  PrefSetAppPreferences (appFileCreator, appPrefID, appPrefVersionNum,
      &prefs, sizeof (prefs), true);
}


/***********************************************************************
 *
 * FUNCTION:    StarterPilotMain
 *
 * DESCRIPTION: This is the main entry point for the application.
 * PARAMETERS:  cmd - word value specifying the launch code.
 *                 cmdPB - pointer to a structure that is associated with the
launch code.
 *              launchFlags -  word value providing extra information about the
launch.
 *
 * RETURNED:    Result of launch
 *
 * REVISION HISTORY:
 *
 *
 ***********************************************************************/
static DWord StarterPilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{
  Err error;


  error = RomVersionCompatible (ourMinVersion, launchFlags);
  if (error) return (error);


  switch (cmd)
      {
      case sysAppLaunchCmdNormalLaunch:
          error = AppStart();
          if (error)
              return error;

          FrmGotoForm(MainForm);
          AppEventLoop();
          AppStop();
          break;

      default:
          break;

      }

  return 0;
}


/***********************************************************************
 *
 * FUNCTION:    PilotMain
 *
 * DESCRIPTION: This is the main entry point for the application.
 *
 * PARAMETERS:  cmd - word value specifying the launch code.
 *                 cmdPB - pointer to a structure that is associated with the
launch code.
 *              launchFlags -  word value providing extra information about the
launch.
 * RETURNED:    Result of launch
 *
 * REVISION HISTORY:
 *
 *
 ***********************************************************************/
DWord PilotMain( Word cmd, Ptr cmdPBP, Word launchFlags)
{
   return StarterPilotMain(cmd, cmdPBP, launchFlags);
}
```

たとえば，「3:00～会議」と入れておけば3:00にDate Bookが起動してアラーム音を出力しますね。

そのほかにも，Palm OSにユニークな機能として「検索」するために起動するというオプションがあります（検索機能については囲み参照）。面白いのはシステムが検索コマンドを持っているのではなく，

システムが検索用ダイアログを表示
検索文字列が入力されると，システムはアプリケーションを端から起動
各アプリケーションが，実際の検索作業を開始
システムは，各アプリケーションの検索結果を表示

という手順にて行われます。文字列の検索中に，アプリケーションの起動画面などがいちいち表示されては，うっとうしいだけですから，検索用に起動された場合は，画面表示を行ってはいけません。

このようにどのような目的でアプリケーションが起動されたかを知るのが起動コードです。Palm OSからアプリケーションが起動される際，起動コードがPilotMain()関数に渡されますので，この値によって，アプリケーションが現在なにを行わなければならないのかを知ることができます。

とりあえずは，デスクトップから起動された場合のsysAppLaunchCmdNormalLaunchのみをサポートしておけばよいでしょう。

ちなみに，sysAppLaunchCmdNormalLaunch以外で起動された場合は，グローバル変数領域は確保されずに起動されます。このことにより「検索」コマンドなどで高速にアプリケーションを切り替えることができますが，うっかりこのルールを忘れて，グローバル変数へアクセスを行うと「アドレスエラー」となりますので注意してください。

＊3；どうも「Pilot」というのは，某万年筆メーカーの登録商標だったようです。商品名自身は，Palmと変わったのですが，技術文書のなかには，まだ「Pilot」という名前が随所に残されています。

イベント

表3にPalm OSで使用される主なイベントの種類をまとめてみました。いわゆるイベントドリブンと呼ばれるOSで用意されているものはひととおり用意されています。

「ペンがコントロール内で押されたが，コントロール外で離された」（例：ctlExitEvent）というようなあまり使わないイベントは，表3から除外してあるのですが，それでも多いと感じるかもしれません。Palm OSではイベント名は，

コントロールの種類（3文字）＋イベントの種類

というルールで命名されているので，これを考慮して表3を見れば無駄に多

表2

コード	型	名前	入出力	内容
sysAppLaunchCmdAlarmTriggered	設定されたアラーム時間の直前に通知される。次のアラーム時間の設定やアラーム音出力など，時間のかからない処理のみ行うことができる。時間のかかる処理は，sysAppLaunchCmdDisplayAlarmオプションで起動された際に行うこと			
	共有体名：SysAlarmTriggeredParamType			
	型	名前	入出力	内容
	DWord	ref	入力	呼び出し側がアラームをセットしたときに設定できる任意の値
	Ulong	alarmSeconds	入力	アラームの設定された時間。1904/1/1午前0時を起源とする秒で表されている
	Boolean	purgeAlarm	出力	ここにFALSEを設定しリターンすることにより本当のアラーム処理（sysAppLaunchCmdDisplayAlram）をキャンセルできる
sysAppLaunchCmdDisplayAlarm	設定されたアラーム時間に通知される。sysAppLaunchCmdAlarmTriggeredと違い，長時間に及ぶ処理を行うことが可能（ユーザからの入力待ちダイアログなど）			
	共有体名：SysAlarmTriggeredParamType			
	型	名前	入出力	内容
	DWord	ref	入力	呼び出し側がアラームをセットしたときに設定できる任意の値
	Ulong	alarmSeconds	入力	アラームの設定された時間。1904/1/1午前0時を起源とする秒で表されている
	Boolean	soundAlarm	入力	TRUEの場合，アラーム音を出力する（現バージョンでは未サポート）
sysAppLaunchCmdFind	グローバル検索（デスクトップの"Find"ボタンを押して起動）のためアプリケーションを起動。なお，検索結果はプログラムの戻り値ではなく，検索ヒットごとにFindSaveMatch()関数を用いシステムに報告する			
	共有体名：FindParamsType			
	型	名前	入出力	内容
	Word	dbAccessMode	入力	検索モード。セキュリティのかかったデータは，ある特定の場合にしか表示しない
	Word	recordNum	入力	前回，ヒットしたデータのレコード
	Boolean	more	出力	なんらかの理由で検索を一時中止した場合，Trueを返す
	Char [ ]	strAsTyped	入力	"Find"ダイアログ中，入力されたデータ
	Char [ ]	strToFind	入力	strAsTypedをすべて小文字にしたもの。大文字/小文字の不一致を検索時に避けたい場合は，こちらを検索文字列として参照すること
sysAppLaunchCmdGoto	検索結果の表示のように，アプリケーション起動後，すぐに特定のデータを表示するためのオプション			
	共有体名：GoToParamsType			
	型	名前	入出力	内容
	Word	searchStrLen	入力	表示するデータの文字列長
	Word	dbCardNo	入力	表示するデータベースのカードナンバー
	LocalID	dbID	入力	表示するデータベースのローカルID
	Word	recordNum	入力	表示するレコードのインデックス
	Word	matchPos	入力	検索で，一致した文字列へ位置
	Word	matchFieldNum	入力	検索で一致した文字列があるフィールドナンバー
	DWord	matchCustom	入出力	アプリケーションで使用可

sysAppLaunchCmdInitDataBase	データベースを初期化するため、アプリケーションを起動　例）Hot Syncから新しいデータがインストールされたときなど
sysAppLaunchCmdNormalLaunch	通常の起動方法
sysAppLaunchCmdPanelCalledFromApp	モデムの設定やネットワークの設定を変えるPreference Panelアプリケーション（Windowsのコントロールパネル内で使用される*.cplアプリケーションに相当）用。デスクトップでなく、ある別なアプリケーションから起動されていることを示す
sysAppLaunchCmdReturnFromPanel	Preference Panelでの作業が終わったので、アプリケーションを再開
sysAppLaunchCmdSystemResetr	システムリセット後、すべてのアプリケーションに通知される　なお、この起動オプションで画面表示を行ってはいけない
sysAppLaunchCmdTimeChange	システム時間が変更されたので、必要なデータを再チェックするためにアプリケーションを起動

表3

イベントの種類	名前	内容		
システム	penDownEvent	ペンが押された		
	penMoveEvent	ペンが動いた（ドラッグした）		
	penUpEvent	ペンが離された		
	keyDownEvent	文字入力エリアに文字が入力された		
		共有体名：keyDown		
		型	名前	内容
		Word	chr	入力された文字のASCIIコード
		Word	KeyCode	入力された文字のキーコード
		Word	modifiers	文字入力時の文字入力モードの状態（シフトモード，Ctrlモード）
	winEnterEvent	ウィンドウをActivateする		
		共有体名：winEnter		
		型	名前	内容
		WinHandle	enterWindow	Activateするウィンドウのハンドル
	winExitEvent	ウィンドウをdeactivateする		
		共有体名：winExit		
		型	名前	内容
		WinHandle	exitWindow	Deactivateするウィンドウのハンドル
メニュー	menuEvent	メニューが選択された		
		共有体名：menu		
		型	名前	内容
		Word	itemID	メニューアイテムのリソースID（コマンドID）
アプリケーション	appStopEvent	アプリケーションの終了要求　例）システムがほかのアプリケーションを起動したいときなど		
フォーム	frmCloseEvent	フォーム（ウィンドウ）を閉じる要求が出された		
		共有体名：frmClose		
		型	名前	内容
		Word	formID	閉じるべきフォームのID
	frmGotoEvent	デスクトップの“Find”などで，特定のデータを表示する要求が出された		
		共有体名：frmGoto		
		型	名前	内容
		Word	formID	表示するべきフォームのID
		Word	recortdNum	表示するレコードのインデックス
		Word	matchPos	表示する文字列の位置
		Word	matchLen	表示する文字列の文字列長
		Word	matchFieldNum	検索で，一致した文字列があるフィールドナンバー
		Dword	matchCustom	アプリケーションで使用可
	frmLoadEvent	フォーム（ウィンドウ）リソースをメモリへ読み込む		
		共有体名：frmLoad		
		型	名前	内容
		Word	formID	読み込むフォームのID
	frmOpenEvent	フォーム（ウィンドウ）リソースを開く		
		共有体名：frmOpen		
		型	名前	内容
		Word	formID	開くフォームのID
	frmSaveEvent	アプリケーションがメモリ上に持っているデータを保存する		
	frmUpdateEvent	画面を再描画する		
		共有体名：frmUpdate		
		型	名前	内容
		Word	formID	再描画するフォームのID
		Word	updateCode	アプリケーションが自由に使用可能
	nilEvent	ヌルイベント（システムが暇なとき、一定間隔ごとに送られてくる）		
		例）アニメーション処理を、１コマずつ進める場合などに使用		

部品	イベント			
ボタン チェックボックス ポップアップトリガー 押しボタン リピートボタン セレクトトリガー	ctlEnterEvent	コントロールの領域内でペンが押された		
		共有体名：ctlEnter		
		型	名前	内容
		Word	controlID	コントロールのID
		void*	pControl	コントロール情報の構造体（ControlType）へのポインタ
	ctlSelectEvent	コントロールの領域内でペンが押されたあと、領域内で離された		
		共有体名：ctlSelect		
		型	名前	内容
		Word	controlID	コントロールのID
		void*	pControl	コントロール情報の構造体（ControlType）へのポインタ
		Boolean	on	ボタンが押された場合に、True。それ以外はFalse
	ctlRepeatEvent	リピートボタン内で、ペンが押されている間、1/2行ごとに発行される		
		共有体名：ctlRepeat		
		型	名前	内容
		Word	controlID	コントロールのID
		void*	pControl	コントロール情報の構造体（ControlType）へのポインタ
		DWord	time	イベントが発生したときのシステム時間（System-ticks）
文字列入力フィールド	fldEnterEvent	コントロール内でペンが押された		
		共有体名：fldEnter		
		型	名前	内容
		Word	fieldID	コントロールのID
		void*	pField	コントロール情報の構造体（FieldType）へのポインタ
フォームタイトル	frmTitleEnterEvent	フォームのタイトル文字列が押された		
		共有体名：frmTitleEnter		
		型	名前	内容
		Word	formD	コントロールのID
	frmTitleSelectEvent	フォームのタイトル文字列が押れたあと、タイトルバー内で離された		
		共有体名：frmTitleSelect		
		型	名前	内容
		Word	formD	コントロールのID
リスト	lstSelectEvent	リストコントロールで、アイテムが選択された		
		共有体名：lstSelect		
		型	名前	内容
		Word	listD	コントロールのID
		void*	pList	コントロール情報の構造体（ListType）へのポインタ
		Word	Selection	選択されたアイテムのID
スクロール	sclExitEvent	スクロールバーからペンが離れた		
		共有体名：sclExit		
		型	名前	内容
		Word	scrollBarID	コントロールのID
		void*	pScrollBar	コントロール情報の構造体（ScrollType）へのポインタ
		Short	Value	以前のスクロールの値
		Short	newValue	新しいスクロールの値
	tblSelectEvent	テーブルコントロール内のアイテムが選択された		
		共有体名：tblSelect		
		型	名前	内容
		Word	tableID	コントロールのID
		void*	pTable	コントロール情報の構造体（TableType）へのポインタ
		Short	row	何行目の項目か？
		Short	Column	何列目の項目か？
ポップアップトリガー	popSelectEvent	ポップアップセレクトのアイテムが選択された		
		共有体名：popSelect		
		型	名前	内容
		Word	controlD	コントロールのID
		void*	controlP	コントロール情報の構造体（ControlType）へのポインタ
		Word	listID	ポップアップリストのリソースID
		void*	listP	ポップアップリスト構造体（ListType）へのポインタ
		Word	selection	新しく選択されたアイテム（いちばん上が０）
		Word	priorSelection	以前に選択されていたアイテム（いちばん上が０）

表4

●イベント処理関数

◎void EvtGetEvent(

```
EventPtr  event,          // Event構造体へのポインタ
Long   timeout)           // イベント待ちタイムアウト指定(単位 10[mSec])
```
機能：イベントキューから次に処理すべきイベントを拾得する
注：timeoutで指定された時間内にイベントが発生しなかった場合にはnilEventが発生する

▌フォームマネジャ

●FormType構造体

```
typedef  struct {
  WindowType  window;          // フォームの使用するウィンドウ
  Word  formID;                // フォームのID
  FormAttrType  attr;          // 通常は1
  WinHandle  bitBehindForm;    // 画面の退避エリア。ダイアログボックスなどで処理終了後、
後ろの画面を復帰させたいときなどに使う
  FormEventHandlePtr  handler; // イベントハンドラ関数へのポインタ
  Word  forcus;                // フォーカスされているコントロールのIndex
  Word  defaultButton;         // デフォルトのボタンのIndex
  Word  menuRscID;             // ヘルプのリソースID
  Word  numObjects;            // フォーム内に存在するオブジェクトの数
  FormObjListType*  objects;   // フォーム内に存在するオブジェクトデータへのリンク
} FormType;
```

●フォーム操作関数

◎Word FrmAlert(

```
Word alertId)             // アラートリソースID
```
機能：指定したアラートをモーダル(Modal)ダイアログとして表示しボタンが押されるまで待つ
戻り値：アラートに定義されたボタン番号

◎void FrmCopyTitle(

```
FormPtr  frm,             // フォーム構造体へのポインタ
CharPtr  newTitle)        // 設定するタイトル
```
機能：フォームタイトルを設定する
戻り値：なし
注：新しいタイトルがフォームリソースで定義された元のタイトルの長さを超えてはならない

◎Word FrmCustomAlert(

```
Word  alertId,            // アラートリソースID
CharPtr  s1,              // ^1に対応する文字列
CharPtr  s2,              // ^2に対応する文字列
CharPtr  s3)              // ^3に対応する文字列
```
機能：指定したアラートを表示しホタンが押されるまで待つ
戻り値：アラートに定義されたボタン番号
注：アラートリソース中のオブジェクトに^1, ^2, ^3という文字列を設定するとアラート表示時にs1, s2, s3文字列と置き換えることができる。文字列を表示したくない場合にはNULLでなく""を渡すこと

◎void FrmDeleteForm(

```
FormPtr  frm)             // フォーム構造体へのポインタ
```
機能：フォームを操作したメモリブロックを解放する
戻り値：なし

◎Word FrmDoDialog(

```
FormPtr  frm)             // フォーム構造体ポインタ
```
機能：ダイアログフォームを表示しボタンが押されるまで待つ
戻り値：ダイアログに定義されたボタン番号

◎void FrmDrawForm(

```
FormPtr  frm)             // フォーム構造体へのポインタ
```
機能：フォーム上のすべてのオブジェクトを描画する
戻り値：なし

◎FormPtr FrmGetActiveForm(void)

機能：カレントフォームのフォーム構造体へのポインタを取得する
戻り値：カレントフォームのフォーム構造体ポインタ

◎SWord FrmGetControlValue(

```
FormPtr  frm,             // フォーム構造体へのポインタ
Word  controlID)          // コントロールのインデックス番号
```
機能：フォーム上の指定されたコントロール(プッシュボタンとチェックボックス)の値を読み取る。コントロールのインデックスはFrmGetObjectIndex()関数にて取得できる
戻り値：1 = on, 0 = 0ff

◎Word FrmGetObjectIndex(

```
FormPtr  frm,             // フォーム構造体へのポインタ
Word  objID)              // IDリソースID
```
機能：フォームオブジェクトインデックスを取得する
戻り値：オブジェクトインデックス

◎void* FrmGetObjectPtr(

```
FormPtr  frm,             // フォーム構造体へのポインタ
Word  objIndex)           // オブジェクトインデックス
```
機能：フォームオブジェクトのポインタを取得する
戻り値：オブジェクトポインタ

◎WinHandle FrmGetWindowHandle(

```
FormPtr  frm)             // フォーム構造体へのポインタ
```
機能：指定したウィンドウハンドルを取得する
戻り値：ウィンドウハンドル

◎FormPtr FrmInitForm(

```
Word  rscID)              // フォームリソースID
```
機能：フォームリソースを読み込み初期化する
戻り値：フォーム構造体へのポインタ

◎void FrmPopupForm(

```
Word  formId)             // フォームリソースID
```
機能：フォームをオーバラップして表示する
戻り値：なし

◎void FrmRcturnToForm(

```
Word formId)              // フォームリソースID
```
機能：カレントフォームをクローズしてオーバラップされたフォームへ戻る
戻り値：なし
注：フォームリソースIDに0を設定すると自動的にオーバラップされた(下側の)フォームへ戻る

◎void FrmSaveAnForms(void)

機能：現在開かれているすべてのフォームへfrmSaveEventを送る
戻り値：なし

◎void FrmSetActiveForm(

```
FormPtr  frm)             // フォーム構造体へのポインタ
```
機能：アクティブフォームを指定する
戻り値：なし
注：すべてのユーザーインタフェイスイベントはアクティブフォームへ入力される。

◎void FrmSetControlGroupSelection(

```
FormPtr  frm,             // フォーム構造体へのポインタ
Byte  groupNum,           // グループ番号
Word  controlID)          // 追加するコントロールのID
```
機能：コントロールを指定されたグループへ追加する。
戻り値：なし
注：n者択一のように複数のコントロールをグループ化しひとつのプッシュボタンのみを選択可能にすることができる

◎void FrmSetControlValue(

```
FormPtr frm,              // フォーム構造体へのポインダ
Word ojbIndex,            // コントロールへのインデックス
short newValue);          // 新たに設定する値
```
機能：指定されたコントロールに値を設定する。
戻り値：なし
注：この関数を実行後コントロールの再描画が行われる。

◎void FrmSetEventHandler(

```
FormPtr  frm,                       // フォーム構造体へのポインダ
FormEventHandlerPtr  handler)       // ハンドル関数ポインタ
```
機能：指令されたフォームにイベントハンドルルーチンを設定する
戻り値：なし

●システム関連操作関数

◎Boolean SysHandleEvent(

```
EventPtr  eventP)         // イベント構造体へのポインタ
```
機能：システムイベント処理。
戻り値：システムイベントイベントが実行されたか否か
注：EvtGetEvent()関数にてイベントメッセージを取得後ユーザー定義の処理ルーチンで処理されないイベントに対してシステムデフォルト処理を行う。詳しくはCode Worriorsが作成するスケルトンプログラムStarter.cを参照

◎int SysRandom(

```
ULong  newSeed);          // 新しい乱数の種(seed)0を指定すると更新しない
```
機能：0からsysRandomMaxまでの乱数を生成する
戻り値：生成された乱数

◎Word SysTicksPerSecond(void)

機能：Palm OS内で使われるシステム時間(ticks)と1秒の比率を返す
例：イベント取得をEvtGetEvent(&event, SysTicksPerSecond() / 2)のように行えば1秒間に2回のnilEventを発生させることができる

●プリファレンス操作関数

◎SWord PrefGetAppPreferences(

```
DWord  creator,           // Creator ID
Word  id,                 // ID番号(複数のプリファレンスを持つ場合)
VoidPtr prefs,            // プリファレンス保存エリアポインタ
Word* prefsSize,          // プリファレンスサイズ
Boolean saved)            // (true－Saved Preferences, false－Current Preferences)?
```
機能：アプリケーションプリファレンスの取得
戻り値：noPreferenceFound：プリファレンスが見つからない
それ以外：成功
注：文字列などの可変長データを含んだプリファランスはprefs = NULL, *prefsSize = 0として一度PrefGetAppPreferences()関数を呼び出し、データ長を調べたうえでもう一度PrefGEtAppPreferences()関数を呼び出すこと

◎void PrefGetPreferences(

```
systemPrderencesPtr p)    // システムプリファレンス保存エリアポインタ
```
機能：システムプリファレンスの取得
戻り値：なし

◎void PrefSetAppPreferences(

```
DWord creator,            // CreatorID
Word id,                  // ID番号(複数のプリファレンスを持つ場合)
SWord version,            // アプリケーションバージョン
VoidPtr prefs,            // プリファレンス保存エリアポインタ
Word prefsSize,           // プリファレンスサイズ
Boolean saved)            // (true－Saved Preferences, false－Current Preferences)
```

機能：アプリケーションプリファレンスの保有
戻り値：なし

ウィンドウマネージャ

●ウィンドウ描画関数

◎void WinDisplayToWindowPT (
SWordP extentX　　// X座標変数へのポインタ
SWordP extentY);　　// Y座標変数へのポインタ
機能：指定されたX,Y座標をディスプレイ座標からウィンドウ座標へ変換する。変換された値はextentX, extendYで指定された変数へ返される
戻り値：なし

◎void WinDrawBitmap (
BitmapPtr　bitmapP,　　// ビットマップへのポインタ
Sword x,　　// x座標
Sword y)　　// y座標
機能：ビットマップを与えられた座標に表示する
戻り値：なし

◎void WinDrawChars (
CharPtr　chars,　　// 文字列へのポインタ
Word　len,

// 文字列のバイト数
SWord x,

// 描画開始点
SWord y)
機能：指定位置に文字列を表示する
戻り値：なし
注：FntSetFont()関数により使用する文字のフォントが指定できる

◎void WinDrawLine (
short x1,
short y1,
short x2,
short y2)
機能：(x1, y1)から(x2, y2)まで線分を描画する
戻り値：なし

◎void WinDrawRectangle (
RectanglePtr　r,　　// レクタングル構造体へのポインタ
Word cornerDiam)　　// 角の丸み(0=直角)
機能：レクタングルで指定された領域内を黒く塗りつぶす
戻り値：なし

◎void WinEraseLine (
short x1,
short y1,
short x2,
short y2);
機能：(x1, y1)から(x2, y2)まで線分を消去する
戻り値：なし

◎void WinEraseRectangle (
RectanglePtr　r,　　// レクタングル構造体へのポインタ
Word cornerDiam)　　// 角のまるみ(0=直角)
機能：レクタングルで指定された領域内を白で消去する
戻り値：なし

◎void WinFillLine (
SWord x1,
SWord y1,
SWord x2,
SWord y2);
機能：(x1, y1)から(x2, y2)までWinSetPattern()関数で指定されたパターンで線分を描画する
戻り値：なし

◎void WinFillRectangle (
RectanglePtr　r,　　// レクタングル構造体へのポインタ
Word cornerDiam)　　// 角の丸み(0=直角)
機能：レクタングルで指定された領域内をWinSetPattern()関数で指定されたパターンで塗りつぶす
戻り値：なし

◎void WinGetDisplayExtent (
SWordPtr extentX,　　// x座標変数へのポインタ
SWordPtr extentY)　　// y座標変数へのポインタ
機能：描画領域(スクリーン)の大きさを返す
戻り値：なし

◎WinPtr WinGetWindowPointer (
Winnandle winHandle)　　// ウィンドウハンドル
機能：ウィンドウ構造体へのポインタを取得する
戻り値：ウィンドウ構造体へのポインタ

◎void WinInvertRectangle (
RectanglePtr　r,　　// レクタングル構造体へのポインタ
Word cornerDiam)　　// 角の丸(0=直角)
機能：指定された領域内の表示を反転させる

◎void WinSetPatetrn (
CustomPatternType pattern);　　// パターン構造体
機能：WinFillLin()WinFillRectangle()関数で使用する塗りつぶしパターンを指定する
注：patternは8バイトからなる8×8の白黒ベタビットパターン

●フォント操作関数

◎FontID FntSetFont (
FontID fonID)　　// フォントID
機能：標準フォントを設定する
戻り値：関数実行前に設定されていた標準フォントID
注：標準システムで使えるフォントIDは以下のとおり。
　stdFont　　通常のフォント
　boldFont　　太字のフォント
　largeFont　　大きいフォント
　largeBoldFont　　太字の大きいフォント

コントロールマネージャ

●Control構造体

```
typedef struct {
Word　id;                      // コントロールのID(通常リソースIDと同一にする)
RectangleType bounds;          // フォーム内でのコントロールの表示領域
Char　Ptr;                     // コントロールのラベル文字列
CntrolAttrType　attr;          // 後述
FondID　font;
Byte　group;
} ControlType;
typedef　ControlType*　ControlPtr;

typedef struct {
Byte　usable:1,                // 通常1に設定する
Byte　enabled:1,               // 0の場合コントロールを灰色表示し無効にする
Byte　visible:1,               // システム内部で使用
Byte　on:1,                    // チェックボックスなどで「on」状態のときに1
Byte　leftAnchor:1,            // ラベル文字列にあわせて大きさを変える際左端で固定させ
                                  たいときに1を設定
Byte　frame:3                  // コントロールの種類など
} ControlAttrType;
```

●コントロール操作関数

◎short　CtlGetValue (
ControlPtr ControlP)　　// コントロール構造体へのポインタ
機能：チェックボックスなどのオブジェクトから値を取得する
戻り値：収得された値(0= off, 1 = on)

◎void CtlSetValue (
ControlPtr ControlP,　　// コントロール構造体へのポインタ
short newValue)　　// 設定する植(0= off, 1 = on)
機能：チェックボックスなどのオブジェクトへ値を設定する
戻り値：なし

●テーブル操作関数

◎void TblDrawTable (
TablePtr　table)　　// テーブル構造体へのポインタ
機能：テーブルの描画
戻り値：なし

◎void TblEraseTable (
TablePtr　table)　　// テーブル構造体へのポインタ
機能：テーブルオブジェクトの削除
戻り値：なし

◎void TblGetBounds (
TablePtr　table,　　// テーブル構造体へのポインタ
RectanglePtr r)　　// レクタングル構造体へのポインタ
機能：テーブルの外枠を指定されたレクタングル構造体へ取得する
戻り値：なし

◎Word TblGetNumberOfRows (
TablePtr table)　　// テーブル構造体へのポインタ
機能：テーブル行数の取得
戻り値：なし

◎ULong TblGetRowData (
TabiePtrtable,　　// テーブル構造体へのポインタ
Word row)　　// 行番号
機能：テーブルの行にセットされた値の取得
戻り値：設定されていた値

◎Word TblGetRowID (
TablePtrtable,　　// テーブル構造体へのポインタ
Word row)　　// 行番号
機能：テーブルの行ID番号の取得
戻り値：ID番号

◎void TblMarkRowInvalid (
TablePtr　table,　　// テーブル構造体へのポインタ
Word row)　　// 行番号
機能：指定した行を無効にする
戻り値：なし
注：実際の動作はTblRedwawTable()関数により再描画され際に反映される。

◎void TblRedrawTable (
TablePtr　table)　　// テーブル構造体へのポインタ
戻り値：なし
注：TblMarkRowInvalid()関数により指定された行を再描画する

◎Boolean TblRowUsable (
TablePtr　table,　　// テーブル構造体へのポインタ
Word row)　　// 行番号

機能：指定した行が使用可能かチェックする
戻り値：true = 使用可false = 使用不可

◎void TblSetColumnUsable（
TablePtr table,　// テーブル構造体へのポインタ
Word row,　// 行番号
Bodean usable)　// 設定する状態(true: 使用可false:不可)
機能：カラムを使用可能にする
戻り値：なし
注：使用不可に設定されると表示されない

◎void TblSetCustomDrawProcedure（
TablePtr table,　// テーブル構造体へのポインタ
Word column,　// 列番号
VoidPtr drawCallback)　// 描画関数ポインタ
機能：列単位にカスタム描画関数を設定する
戻り値：なし
注：コールバック関数は以下の引数を持つこと
void TableDrawItemFuncType（
VoidPtr table,
Word row
Word column,
RectanglePtr bounds）;

◎void TblSetItemStyle（
TablePtr table,　// テーブル構造体へのポインタ
Word row,　// 行番号
Word column,　// 列番号
TableItemStyleType type)　// スタイル
機能：テーブル内の指定された項目(row, column)のスタイルを設定する
注：項目スタイルの指定は以下より選択可能
checkboxTableItem?
customTableItem?
dateTableItem?
labelTableItem
numericTablehem
popupTriggerTableItem
textTableItem
textWithNoteTableItem
timeTableItem
narrowTextTableItem

◎void TblSetRowData（
TablePtr table,　// テーブル構造体へのポインタ
Word row,　// 行番号
ULong data)　// 保有データ
機能：行にデータを設定する
戻り値：なし

◎void TblSetRowID（
TablePtr table,　// テーブル構造体へのポインタ
Word row,　// 行番号
ULong data)　// 保存データ
機能：行にIDを設定する
戻り値：なし

◎void TblSetRowUsable（
TablePtr table,　// テーブル構造体へのポインタ
Word row,　// 行番号
Boolean usable)　// 使用可 (true)/不可 (false) を設定
戻り値：なし

▮フィールドマネージャ

●FieldType構造体

```
typedef struct {
  Word id;                    // ID
  RectangleType rect;         // フィールドの表示領域
  FieldAttrType attr;         // さまざまな属性情報(下記参照)
  CharPtr text;               // テキストを収めたポインタ
  VoidHandle textHandle;      // テキストを収めたハンドル(通常はtextを使用)
  LineInfoPtr lines;          // 行データ
  Word textLen;               // テキストの文字数
  Word textBlockSize;         //
  Word maxChars;              // 最大文字数
  Word selFirstPos;           // 選択領域の開始位置
  Word selLastPos;            // 選択領域の終了位置
  Word insPtXPos;             // カーソル位置
  Word insPtYPos;             // カーソル位置
  FontID fontID;              // フォントID
} FieldType;
```

注：これらのメンバ変数を直接アクセスしてはいけない。対応するアクセス関数を経由して操作を行うこと

```
typedef struct {
  Word usable:1;              // 通常1を設定。
  Word visible:1;             // 表示可能か？
  Word editable:1;            // 編集可能か？
  Word singleLine:1;          // 単行のみか？ (0=複数行サポート)
  Word hasFocus:1;            // 現在フォーカスされているか？
  Word dynamicSize:1;         // 高さが可変か？
  Word insPtVisible:1;        // カーソルを表示するか？
  Word dirty:1;               // 編集されたか？
  Word underline:2;           // 下線を引くか？ (noUnderline / grayUnderline /
                              //   solidUnderline)
  Word justification:2;       // 右寄せ/左寄せ  (leftAlign / rightAlign)
  Word autoShift:1;           // Graffitiの自動シフト機能を使用するか？
  Word hasScrollBar:1;        // スクロールバーを持っているか？
  Word numeric:1;             // 数値のみか？
} FieldAttrType;
```

●フィールド操作関数

◎void FldDelete（
FieldPtr fld,　// フィールド構造体へのポインタ
Word start,　// 削除範囲開始位置
Word end)　// 削除範囲終了位置
機能：フィールド内の文字列を削除
戻り値：なし

◎Boolean FldDirty（
FieldPtr fld)　// フィールド構造体へのポインタ
機能：フィールドが更新されたかどうかをチェック
戻り値：更新されたか否か？

◎void FldDrawField（
FieldPtr fld)　// フィールド構造体へのポインタ
機能：フィールド文字列の描画
戻り値：なし

◎void FldGetAttributes（
FieldPtr fld,　// フィールド構造体へのポインタ
FieldAttrPtr attrP);　// 検索結果を返すFieldAttrType構造体へのポインタ
機能：フィールドの属性を調べる
戻り値：なし

◎Handle FldGetTextHandle（
FieldPtr fld)　// フィールド構造体へのポインタ
機能：フィールドの文字列を収めたハンドルを取得
戻り値：テキストハンドル(ハンドルがない場合NULL)

◎Word FldGetTextLength（
FieldPtr fld)　// フィールド構造体へのポインタ
機能：フィールド内文字列の長さを取得
戻り値：文字列長さ

◎CharPtr FldGetTextPtr（
FieldPtr fld)　// フィールド構造体へのポインタ
機能：フィールド文字列へのポインタ

◎Boolean FldInsert（
FieldPtr fld,　// フィールド構造体へのポインタ
CharPtr insertChas,　// 挿入する文字列へのポインタ
Word insertLen)　// 挿入する文字列の長さ
機能：フィールドへ文字列を追加する
戻り値：挿入は成功したか否か？
注：挿入位置はFldSetInsPtPosition()で指定する

◎void FldRecalculateField（
FieldPtr fld,　// フィールド構造体へのポインタ
Boolean redraw)　// 再描画を行う (true) か否か (false)
機能：フィールドの描画情報等を再設定し必要なら再描画する
戻り値：なし

◎void FldSetInsPtPosidon（
FieldPtr fld,　// フィールド構造体へのポインタ
Word pos)　// 挿入キャラクタの位置
機能：キャラクタ挿入位置の設定
戻り値：なし
注1：FldSetInsertionPoint()はカーソルを移動しない。
注2：実際の挿入はFldInsert()関数で行う

◎void FldSetTextHandle（
FieldPtr fld,　// フィールド構造体へのポインタ
Handle textHandle)　// 文字列を含むテキストハンドル
機能：文字列を含むハンドルをフィールドにセット
戻り値：なし

●メニュー操作関数

◎Boolean MenuHandleEvent（
MenuBarPtr MenuP,　// メニューバーポインタ (NULL:カレントメニュー)
EventPtr event,　// イベント構造体ポインタ
WordPtr error)　// エラーコードを得る変数のポインタ
戻り値：メニュー動作が実行されたか否か？

●メモリ操作関数

◎Err MemHandleFree(
VoidHand h)　// メモリハンドル
機能：メモリハントルを解放する
戻り値：実行結果(0：正常終了0以外：エラー)

◎VoidHand MemHandleNew（
ULong size)　// メモリサイズ
機能：ヒープメモリから指定サイズのメモリを確保する
戻り値：実行結果コード(0：正常終了0以外：エラー)

◎ULong MemHandleSize（
VoidHand h)　// メモリハンドル
機能：指定されたメモリハンドルの大きさを取得する。
戻り値：メモリサイズ

◎VoidPtr MemHandleLock（
VoidHand h)　// メモリハンドル

機能：メモリハンドルをロックして実体のメモリポインタを取得する
戻り値：メモリポインタ
注：メモリハンドルはある条件下にて配置（アドレス）が変わる。そのためメモリハンドルをロックし再配置を禁止してから内容を参照しなければならない

◎LocalID　MemHandleToLocalID（
Voidhand h）　// メモリハンドル
機能：メモリハンドルからメモリのローカルIDを取得する
戻り値：ローカルメモリID。取得に失敗した場合は0を返す

◎Err MemHandleUnlock（
VoldHand h）　// メモリハンドル
機能：メモリハンドルのアンロックする。以後メモリの再配置が発生する可能性がある
戻り値：実行結果コード（0：正常終了0以外：エラー）

◎VoidPtf　MemLocalIDToLockedPtr（
LocalID　local,　// ローカルメモリID
UInt　cardNo）　// メモリカード番号
機能：ローカルIDで指定されたハンドルをロッグしメモリポインタを取得する
戻り値：メモリポインタ

●文字列操作関数

◎Int　StrAToI（
CharPtr str）　// 変換する文字列
機能：ASCII文字列を数値に変換する
戻り値：変換した数値

◎CharPtr StrCat（
CharPtr dst,　// 元文字列へのポインタ
CharPtr src）　// 追加する文字列へのポインタ
機能：指定した変数に文字列を追加する
戻り値：元文字列へのポインタ

◎CharPtr StrChr（
CharPtr str,　// 対象となる文字列へのポインタ
Int chr）　// 検索する文字
機能：文字列内から指定された文字を検索する
戻り値：検索された文字を示すポインタ。見つからなかった場合はNULLを返す

◎CharPtr　StrCopy（
CharPtr dst,　// 文字列へのポインタ
CharPtr src）　// コピー先へのポインタ
機能：指定された文字列をコピーする
戻り値：元文字列へのポインタ

◎CharPtr StrIToA（
CharPtr s,　// 出力先を指定するポインタ
Long i）　// 変換する数値
機能：数値を文字列へ変換する
戻り値：変換文字列出力先のポインタ

◎UInt StrLen（
CharPtr src）　// 文字列へのポインタ
機能：文字列の長さ（バイト数）を取得する
戻り値：文字列の長さ

◎CharPtr StrNCopy（
CharPtrdSt,　// 元文字列へのポインタ
CharPtrsrc,　// コピー先のポインタ
Word n）　// コピーするバイト数
機能：文字列を指定バイト数コピーする
戻り値：元文字列へのポインタ

◎SWord StrVPrintF（
CharPtr s,　// 出力先のポインタ
const Char*formatStr,　// フォーマット文字列へのポインタ
VoidPtr　argParam）　// フォーマット文字列で指定されたデータリストへのポインタ
機能：ANSI-Cのvprintf()関数のように与えられたフォーマットにしたがって引数を文字列として出力する
戻り値：フォーマット返還後の出力した文字数。エラーが発生した場合負の値を返す

Palm OSの「検索」

Palm OSには，検索イベントというWindowsにもMacintoshにもSX-Windowにもない，ユニークなイベントがあります。

従来のシステムでは，検索コマンドはアプリケーションごとに持っていて，たとえば，ワープロソフトは，文章の中の特定のキーワードを検索できますが表計算ソフトのファイルの中身を検索することができません。そういえば，原稿締切の一覧を表にまとめたようなぁ，Oh!Xの締め切りっていつだったっけ？　といってExcelの最近使ったファイルを片っ端から検索しても，実は，Wordの罫線機能で作っていたりするとまったく見つかりません。

あるいは，エクスプローラのツール→検索，コマンドで，ファイルを１つひとつ調べることが可能ですが，キーワードがテキスト情報として，ファイルに含まれていなければなりません。

このようにWindowsのキーワード検索は検索されるファイルとそれを作成したアプリケーションとが結びついていないと意味を成しません。

一方，Palm OSでは，システムから検索イベントが各アプリケーションに届いた場合，自分の管理しているデータのなかにキーワードがあるか否かを調べてシステムに報告します。別のいい方をすると，システムがキーワード検索を各アプリケーションに行わせるので，検索時にどのようなフォーマットのファイルでも検索対象となります。

Date Bookに，Oh!Xの締め切りを書き留めておいても，単なるテキストのメモとして書き留めておいても，To Do Listに書き留めていても，「Oh!X」というキーワード検索で締め切りがわかるわけです。

この検索ボタンを押すと

「Find」と書かれたシステムダイアログが表示されるので「Oh!X」と入力すると，

すべてのアプリケーションが，自分が所有するデータに対し「Oh!X」というキーワードで検索を行う。

いわけではないということがわかります。コントロールの種類はリスト（lst），ポップアップリスト（pop），テーブル（tbl），スクロール（scl），その他（ctl）と５種類しかありません。イベントの種類も，特殊なものを除けば，

EnterEvent　ペンがコントロールに触れた
SelectEvent　ペンがコントロール内で離された
ExitEvent　ペンがコントロール外までドラッグされたあと離された

の3種類しかありません。残りのイベントは特殊な処理を行う場合に参考資料を見ていけばよいでしょう。

「Hello World!!」プログラムを作る

Palm OSの基本がわかったところで，開発環境のCode Warrior for Palm OSにも慣れておきましょう。まずは，画面に「Hello World !!」と表示させるだけのプログラムを作ってみます。

1）Code Warriorsを起動する

スタートメニューから「Code Warriors for Palm OSs」→「Code Warrior IDE」を選択する。

2）プロジェクトを生成する

Code Warriors IDEの「File」→「New...」メニューを選択。

「Project」タブが選択されているのを確認。

Projectの種類は，「Palm OS 3.1」を選択（図3）。

Project nameは，なんでもよいのですが，ここでは「Hello」としておきましょう。

Locationは，ハードディスクの空いているディレクトリを選択してください。

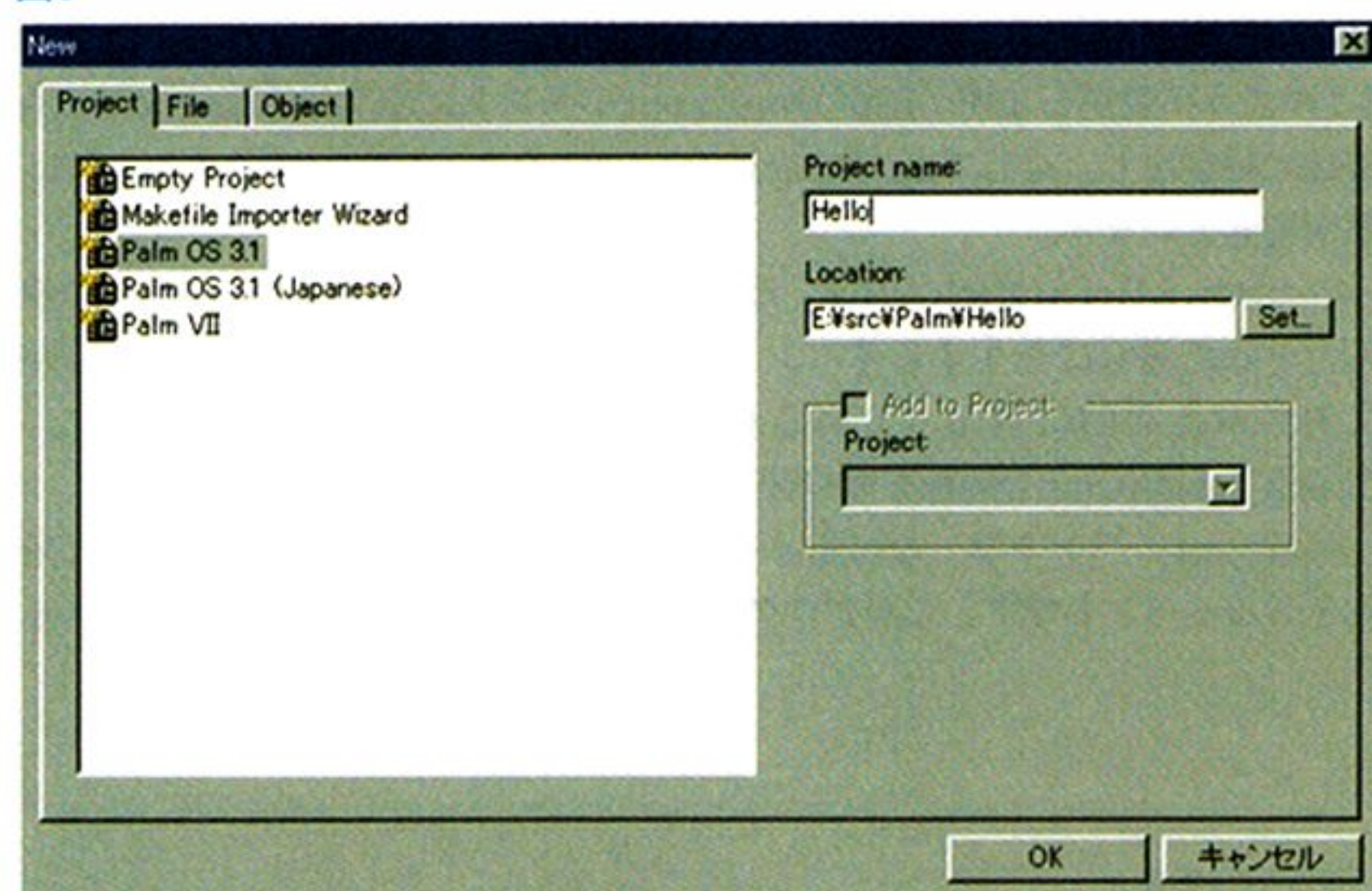
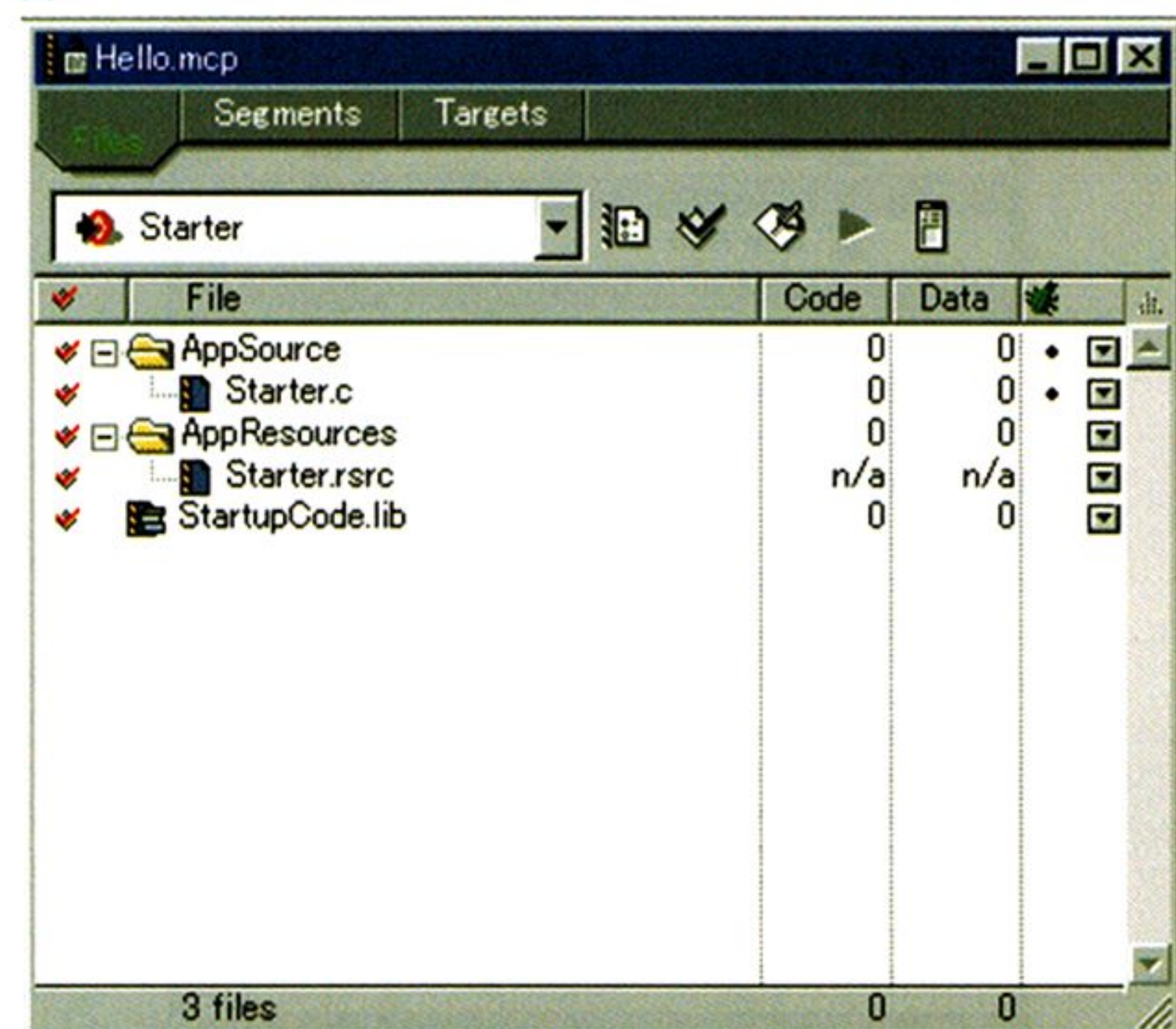
OKボタンを押すと，図4のようなプロジェクト管理ウィンドウが表示されます。

3）プログラムの編集

プロジェクト管理ウィンドウに表示されている「Starter.c」が先ほど説明したスケルトンプログラムです。ここで起動時のOSバージョンのチェックやデフォルトのイベント処理など，Palmアプリケーションに必要な操作を行います。この雛形に対し，アプリケーション固有の動作を加える（今回なら，画面に「Hello World !!」と表示させる）というのがPalmアプリケーションの開発スタイルです。

- プロジェクト管理ウィンドウの中から，「Starter.c」をダブルクリックする。

このファイルのなかで「MainFormHandleEvent (EventPtr eventP)」という関数が，Palm OSからリクエストされるさまざまな要求を対応する処理ルーチンへ振り分けます。デフォルトで，メニュー処理（menuEvent）とフォームのオープン処理（frmOpenEvent）が入っているのを確認してください。

- 関数MainFormHandleEventを編集する。

リスト2のように関数MainFormHandleEventを変更し「Hello World!!」と画面に表示するルーチンを足してください。WinDrawChars()は画面に文字列を表示する関数で，文字列，文字数，表示するX座標，Y座標という引数を取ります。*1

- プログラムをビルドする。

「Project」→「Make」メニューを選択します。運が悪いと「Link Error : Could not open file :e:¥src¥palm¥hello¥Resource.frk¥Starter.tmp」といって，リンカエラーになるかもしれません。おそらく，CWLのバグ（というか仕様？）だと思うのですが，そんなときは「Edit」→「Starter Settings...」で「Starter Settings」ダイアログを開き，左端の項目一覧から，Linker / PalmRez Post Linkerを選択し，右端の「Mac Resource files」をStarter.tmpから，Helloに変更してください（拡張子なし）。せっかくですので，出力ファイルもデフォルトのstarter.prcからHello.prcと変更します（図5）。

ちなみに設定が終わったら，右下の「Save」ボタンで設定を保存し，右上のクローズボタンでダイアログを閉じます。WindowsではなくMacを使っているのだという気構えでいきましょう。

ここで，ビルドが終了したら，作業ディレクトリには「Hello..prc」というファイルが作成されているはずです。これがPalm OSの実行ファイルになります。

- プログラムの実行

実際にPalmマシンを持っている方は「Hello..prc」をPalm Desktopで転送

図3

図4

リスト2

```
/*********************************************************************
 *
 * FUNCTION:    MainFormHandleEvent
 *
 * DESCRIPTION: This routine is the event handler for the
 *              "MainForm" of this application.
 *
 * PARAMETERS:  eventP  - a pointer to an EventType structure
 *
 * RETURNED:    true if the event has handle and should not be passed
 *              to a higher level handler.
 *
 * REVISION HISTORY:
 *
 *
 *********************************************************************/
static Boolean MainFormHandleEvent(EventPtr eventP)
{
    Boolean handled = false;
    FormPtr frmP;


   switch (eventP->eType)
       {
       case menuEvent:
           return MainFormDoCommand(eventP->data.menu.itemID);

       case frmOpenEvent:
           frmP = FrmGetActiveForm();
           MainFormInit( frmP);
           FrmDrawForm ( frmP);
           WinDrawChars("Hello World !!", 14, 10, 30);
           handled = true;
           break;

       case frmUpdateEvent:
           frmP = FrmGetActiveForm();
           FrmDrawForm( frmP);
           WinDrawChars("Hello World !!", 14, 10, 30);
           handled = true;
           break;
       default:
           break;

       }

   return handled;
}
```

し，実行させてください。あるいは，Palm OS Emulator（POSE）&ROM Imageをすでにインストール済みの場合は（Code Warriors Liteではなく）POSEのメニュー（右クリックで表示されます）「Install Application / Data Base ...」→「Other...」から，先ほどのHello..prcを選択してください。

Palm OSのデスクトップ上に新たに「Hello.」というアイコンが登録されているはずです。これをクリックしてください（図6）。

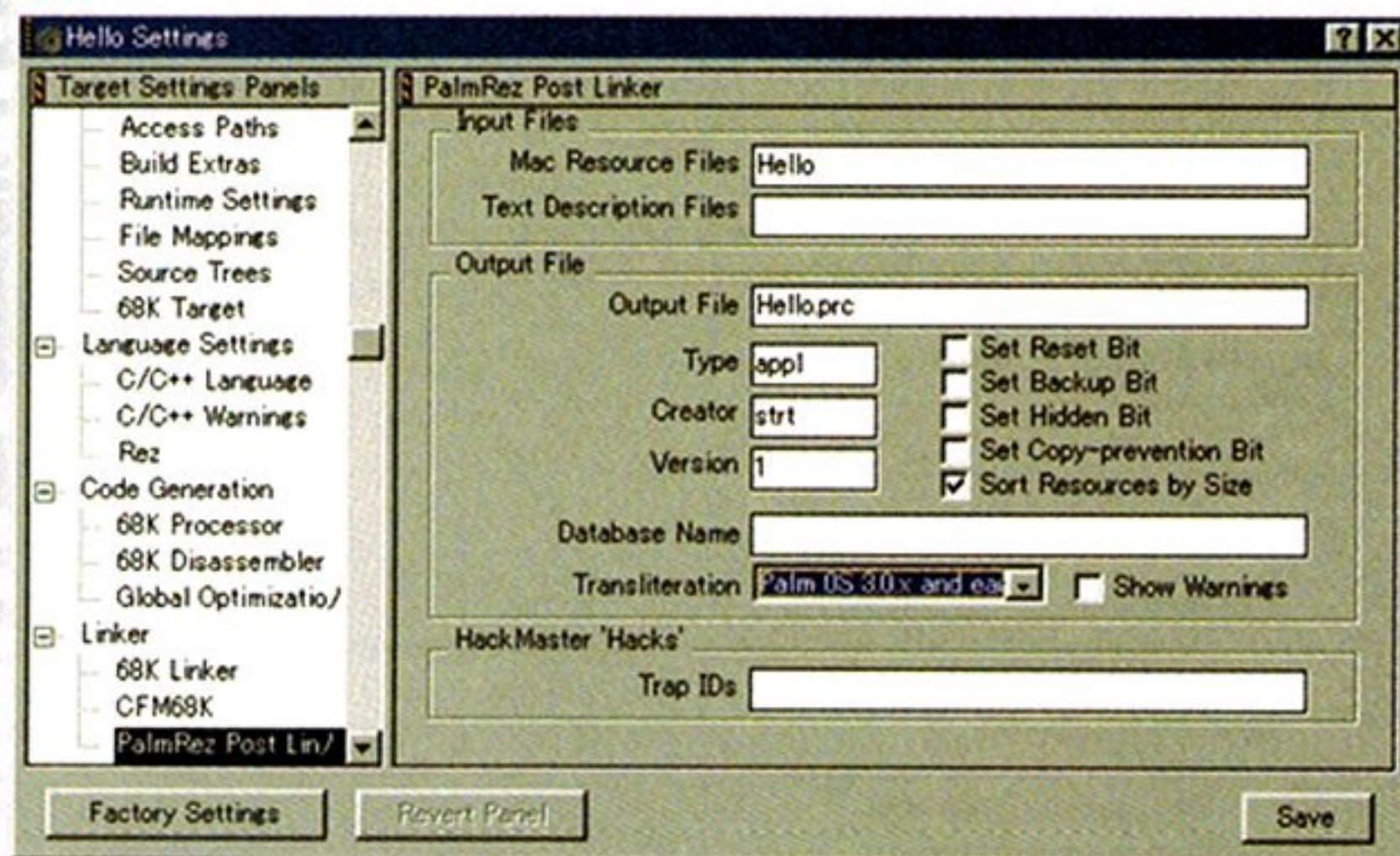
このプログラムは，Code Warriorsの評価版で作られたものであり商用目的には配布してはならないという趣旨のダイアログが表示されますが，納得のうえ「OK」ボタンを押してください。

はい。正しく画面に「Hello World !!」と表示されましたね（図7）。Palm OSではアプリケーションの終了という概念が（エンドユーザーに向けては）ないので，デスクトップ画面に戻す場合は左下にある家のアイコンをクリックします。

アプリケーション終了時に「SystemMgr.c」の4384行目でメモリリークしている可能性があるとPOSEが警告してくるかもしれませんが，これはランタイムルーチン内の話で，我々にとってはどうしようもない部分ですので，とりあえず無視しましょう。

*4：2カ所で文字列を表示させているのは，
frmOpenEventは，アプリケーションが最初に起動されたとき
frmUpdateEventは，上に重なっていた別のフォームが閉じられ，新たに自分のフォームを表示するとき
と2種類のイベントに対応させているからです。通常は描画部分をサブルーチンとして共有するのですが，今回は1行のみですので，2つのイベント処理部分に別々に埋め込んでいます。frmOpenEvent処理部分のbreakを外してそのままfrmUpdateEventに突入させても動くのですが，作法として美しくないのでやってません。

Graffiti練習プログラム

Palmマシンを使っていていちばん困ることは，大事なときに電池がよく切れることですが，2番目に困るのはあの独自の文字入力システムGraffitiです。私は長年シャープのWizを使ってきたので，「∧」→「A」，「┌」→「F」，「┐」→「T」というルールにはなかなか馴染めませんでした。

最近，ケンシロウがPCのキーボード練習を手伝ってくれるソフトや，ゾンビがアルファベットぶら下げて攻めてくるのをキーボード入力で退治するゲームなどを見て，誰がいったい買うのだろうと不思議に思っていましたが，ふむふむ，私がGraffiti用に必要になってしまいました。Palmアプリケーション集CD-ROMなどを探せば，Graffiti練習ソフトなどはいくらでもあるのでしょうが，せっかくの機会ですのでPalmプログラミングと一緒にGraffitiも練習してしまいましょう。

市販ソフトを作るわけではありませんので，あまり凝ったことはしません。画面右隅からアルファベットが延びてくるので，同じ文字をGraffitiで入力して撃退します。1度に複数のアルファベットを撃退すると，2文字目＝2点，3文字目＝3点と小ボーナスが得られます。目標文字列が画面左端まで到達してしまうか100文字画面に出現するとゲーム終了です。

先ほどと同様にプロジェクトを作成してください。ただし，プロジェクト名を「GraffitiPrc」としておいてください。

そして，グローバル変数として，以下の3つを宣言しておいてください。場所はどこでもよいのですが，スケルトンプログラムに"Global Variables"とコメントされたエリアがありますので，その直後に置くのが妥当でしょう。

```
char    szTarget[13];
int     iLeft, iScore, iMissed;
Boolean         bGameOver;
```

名前のとおり，上から，的になる文字列（1画面中に13文字まで），残りの文字列，スコア，ミスした回数，ゲームオーバーになったかという変数です。

次に，スケルトンプログラムのMainFormHandleEvent()関数をリスト3-1のように変更します。frmOpenEventイベント対応部分にGameStart()関数の呼び出し，そしてkeyDownEventイベント，nilEventイベント対応部が増えています。

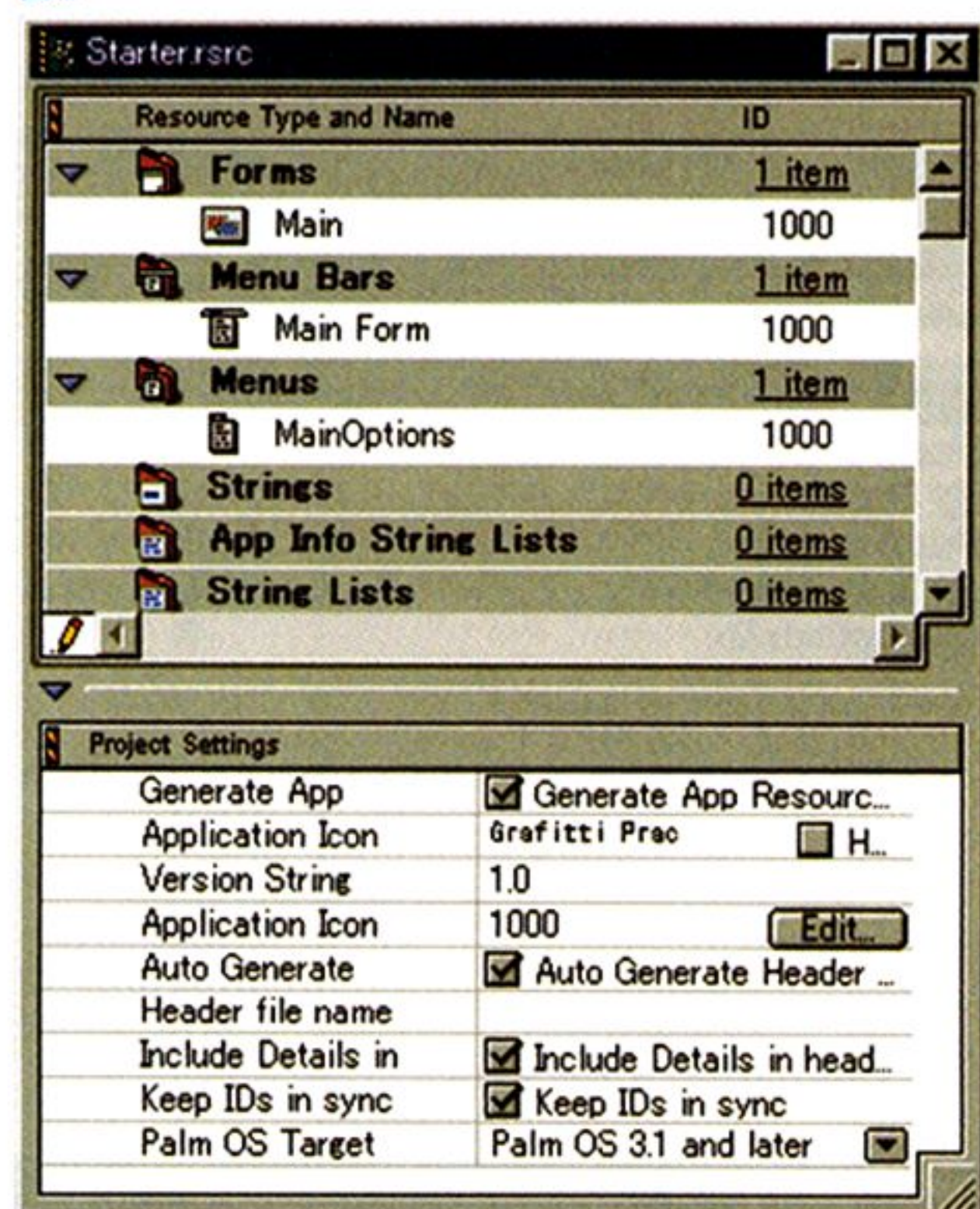
keyDownEventイベントは，その名のとおり，Graffitiでなにか文字が入力された場合に発生するイベントです。このプログラムの場合，文字が入力されたら的に当たっているかを判定しなければなりませんので，CheckTargetChar()関数を呼び出しています。

nilEventイベントは，なにもイベントが発生しないときに，恣意的に発生させるイベントです。このプログラムでは，一定時間ごとにアルファベッ

図5

図6

図7

図8

トが攻めてきますので，そのタイミング生成に使用しています。

このnilEventイベントはデフォルトでは発生しません。そこでスケルトンプログラム中，AppEventLoop()関数をリスト3-2のように変更します。前述のようにbGameOverという変数はゲームオーバーか否かを表しています。ゲームオーバーのときは(ゲーム中でないときは)，アルファベットを生成する必要がありませんのでデフォルトと同じ処理を行っています。そうでないとき(ゲーム中)は，

EvtGetEvent (&event, SysTicksPerSecond () ＊ 10 / 7);

と，2番目の引数を変更しています。これは「何Tips，イベントが発生しなければnilEventを発生するのか」を指定する引数です。TipsというのはPalm OSの時間の単位で，私のVisor Deluxという機械では，1 Tips = 100 [mSec]のようです。ただし，今後，より高速なCPUを持った新型機が登場した場合は，この関係は保証されません。プログラムが機械のスピードに関係なく，将来も問題なく使えるようにするには，

EvtGetEvent (&event, 70);

のように定数を用いてはいけません。

システム関数SysTicksPerSecond()は「現在プログラム実行中のPalm マシンはどのくらい速いのか」を「1秒あたり何Tips生成するか」として返す関数ですので，リスト3-3のように用いればPalmマシンのCPUスピードに関係なく一定時間ごとにnilEventを生成できるようになります。

言葉で説明するとややこしいですが，

EvtGetEvent (&event, SysTicksPerSecond ());

で，1秒間に1回だけnilEventイベントを生成します。これよりも頻度を上げたい場合には，SysTicksPerSecondに1より小さい値を掛けます。頻度を下げたい場合には SysTicksPerSecondに1より大きい値を掛けます。

実際は，Cold Fireは，浮動小数点演算機能を持っていないため，専用ライブラリをリンクするか，システムコールを呼び出すかと，手間がかかりますので(＊ 10 / 7)のように整数でなんとか辻褄をあわせます。

あとは，実際にゲームを開始したり，的を作ったり，当たり判定を行ってたりする関数をスケルトンプログラムの最後尾にでも追加しておいてください(リスト3-4)。その際，プロトタイプ宣言も忘れずに。

これでビルドすればプログラムは完成なのですが，このままではPalm OSのデスクトップに表示される文字とアイコンが先ほどの「Hello World !!」のときと同じStarterですので，これも変更します。

まず「Code Warriors」プロジェクト管理ウィンドウのApp Resources / Starter.rsrcをダブルクリックしてください。図9のような横に細長いウィンドウと普通のウィンドウの2つが現れたはずです。Windowsを使い慣れた感覚から見ると特異に感じるのですが，おそらくMacintoshの流儀なのでしょう。Macintoshのアプリケーションはメニューはメインウィンドウに持たせず，Windowsでいうところのツールバー(「スタート」ボタンやIMEのアイコンが表示されるところ)に表示させることになているので，きっとこうなったのでしょう。それはともかくプロパティウィンドウの2段目Application Iconのところを，Graffiti Practiceと変更し，4段目

リスト3-1

```
static Boolean MainFormHandleEvent(EventPtr eventP)
{
    Boolean handled = false;
    FormPtr frmP;

   switch (eventP->eType)
       {
       case menuEvent:
           return MainFormDoCommand(eventP->data.menu.itemID);

       case frmOpenEvent:
           frmP = FrmGetActiveForm();
           MainFormInit( frmP);
           FrmDrawForm ( frmP);
           GameStart();
           handled = true;
           break;
       case keyDownEvent:
       {
           Char    theChar = eventP->data.keyDown.chr;
           if(CheckTargetChar(theChar)) {
               OnDraw();
           }
           handled = true;
       }
           break;
       case nilEvent:
           AddTargetChar();
           OnDraw();
           handled = true;
           break;
       default:
           break;

       }

   return handled;
}
```

リスト3-2

```
static void AppEventLoop(void)
{
   Word error;
   EventType event;

   do {
       if(bGameOver)
           EvtGetEvent(&event, evtWaitForever);
       else
           EvtGetEvent(&event, EvtGetEvent(&event, SysTicksPerSecond() * 10 /
7);

       if (! SysHandleEvent(&event))
           if (! MenuHandleEvent(0, &event, &error))
               if (! AppHandleEvent(&event))
                   FrmDispatchEvent(&event);

   } while (event.eType != appStopEvent);
}
```

図9

図10

図11

Palm OS でのメモリ

プログラム構成そのものとは関係ありませんが，Palm OSに特有の問題として，メモリの扱いがあります。Palm OS上のプログラムは，

a) グローバル変数　6KB
b) システムの動的割り当て　50KB
c) スタック(ローカル変数含む)　4KB
d) ユーザーの動的割り当て　36KB

しかメモリを用いることができません(Ver 3.xの場合)。b)のシステムの動的割り当てとは，TCP/IPやIrDAなど特定のAPIを呼び出すと割り当てられるメモリのことで，それらの処理が終われば解放されます。

通常のC言語でプログラムむ場合，(ランタイムルーチンを含めて)ユーザープログラムで使用できるメモリは，a)のグローバル変数6KBとc)のローカル変数４KBのみとなってしまいます。これ以上のメモリを扱う場合には，d)のユーザーの動的割り当てと使用します。

まず，この領域からメモリを確保するには，

```
VoidHand        MemHandleNew (Ulong size)
```

関数を用います。ちなみに，VoidHandとはCW LiteのIncludeディレクトリ中common.h内で，

```
typedef void＊       VoidPtr;
typedef VoidPtr＊    VoidHan;
```

のように定義されています(つまりtypedef void ＊＊ VoidHandですね)。

通常，C言語でメモリ領域を確保した場合，新たに確保されたメモリ領域の最初のアドレスへのポインタが得られます(malloc()関数など)。ところが，Palm OSのメモリ確保関数はポインタへのポインタ(void ＊＊)を返します。＊1

たとえば，

```
VoidHand        hMemory = MemHandleNew(100);
```

とすれば，100バイトのメモリ領域が確保されるのですが，

(＊hMemory) でそのメモリ領域の先頭アドレス
(＊hMemory)[0]で，そのメモリ領域の１番目のデータ
(＊hMemory)[99]で，そのメモリ領域の100番目のデータ

をアクセスすることができます。ポインタが直接得られないので，不便に感じるかもしれませんが，これはPalm OSの限られたメモリ空間を有効に使用するための工夫で，以下のような場合に役立ちます。

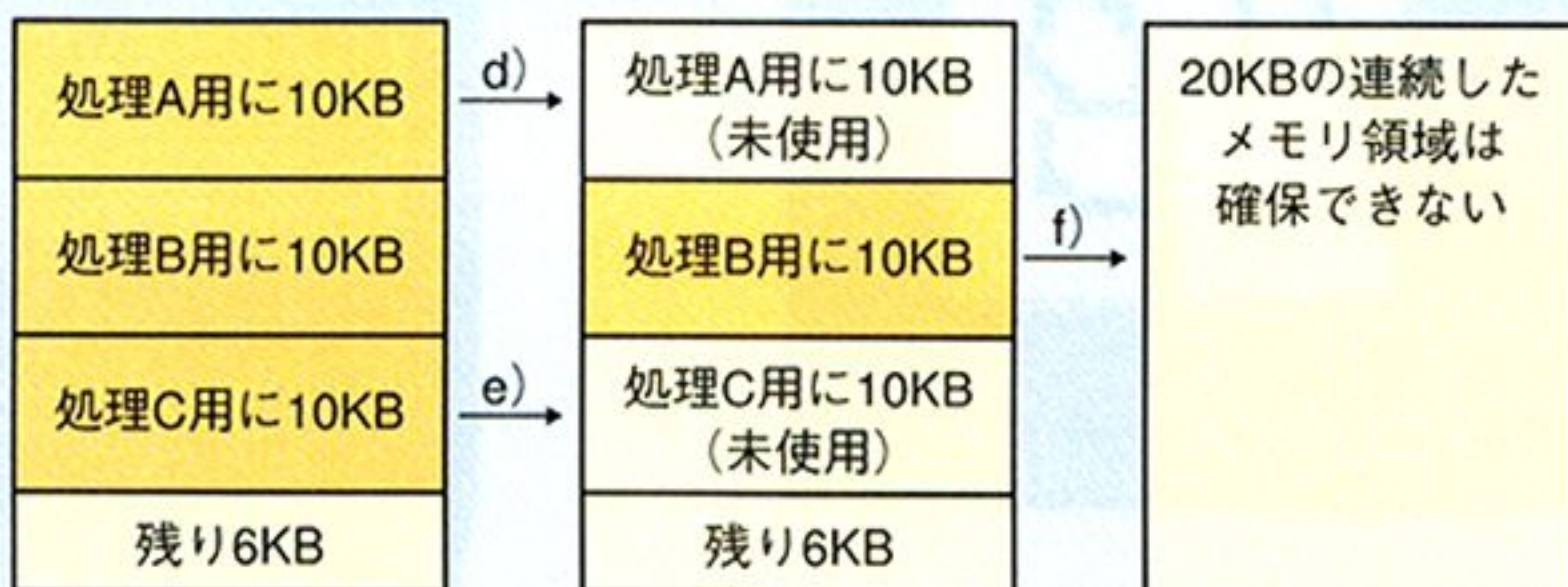
a) 処理Aにて，ワークエリアとして10KBを確保
b) 処理Bにて，ワークエリアとして10KBを確保
c) 処理Cにて，ワークエリアとして10KBを確保
(残り：36 - 10 - 10 - 10 = 6KB)
d) 処理Aが終わり，ワークエリアを解放できる状態に
e) 処理Cが終わり，ワークエリアを開放できる状態に
f) 処理Dにて，ワークエリア20KBを確保

ここで，メモリ領域確保APIがポインタを返してきた場合，図2-1のようになって，f)で必要なメモリ領域が確保できません。e)が終わった時点で無理やり，処理B用のワークエリアを20KBほど前に移動するということもできますがスマートな方法ではありません。

Palm OSではメモリ領域確保APIがハンドルを返すので，図2-2のように，e)の時点で処理B用のワークアリアを移動して，

36 - 20 - 10 + 20 = 16KB

の連続領域を確保できるようになります。

図2-1　メモリ領域をポインタでアクセスする場合

ただし，このようなアクセスで気をつけなければいけないのは，図2-2のようなメモリブロックの移動が起きている最中に，そうとは知らずにアプリケーション側で蛻(もぬけ)のメモリブロック跡をアクセスしてしまうことです(SX-Windowsでは泣かされましたね)。たとえば，

```
VoidPtr           pMemory = ＊hMemory;
for (I = 0; I < 10; I++) {
    PutChar(pMemory[I]);
}
```

と，プログラムが読みやすいようにメモリハンドルの中身をいったんポインタ変数へ移動してループ処理を行っても多くの場合問題ありません。ただし，なにかの都合でループ中にhMemoryの中身が移動した場合，pMemory = ＊hMemoryではなくなってしまいますので，そのあとのループ処理では，まったくデタラメな文字を表示してしまうことにまります。これは防ぐには，

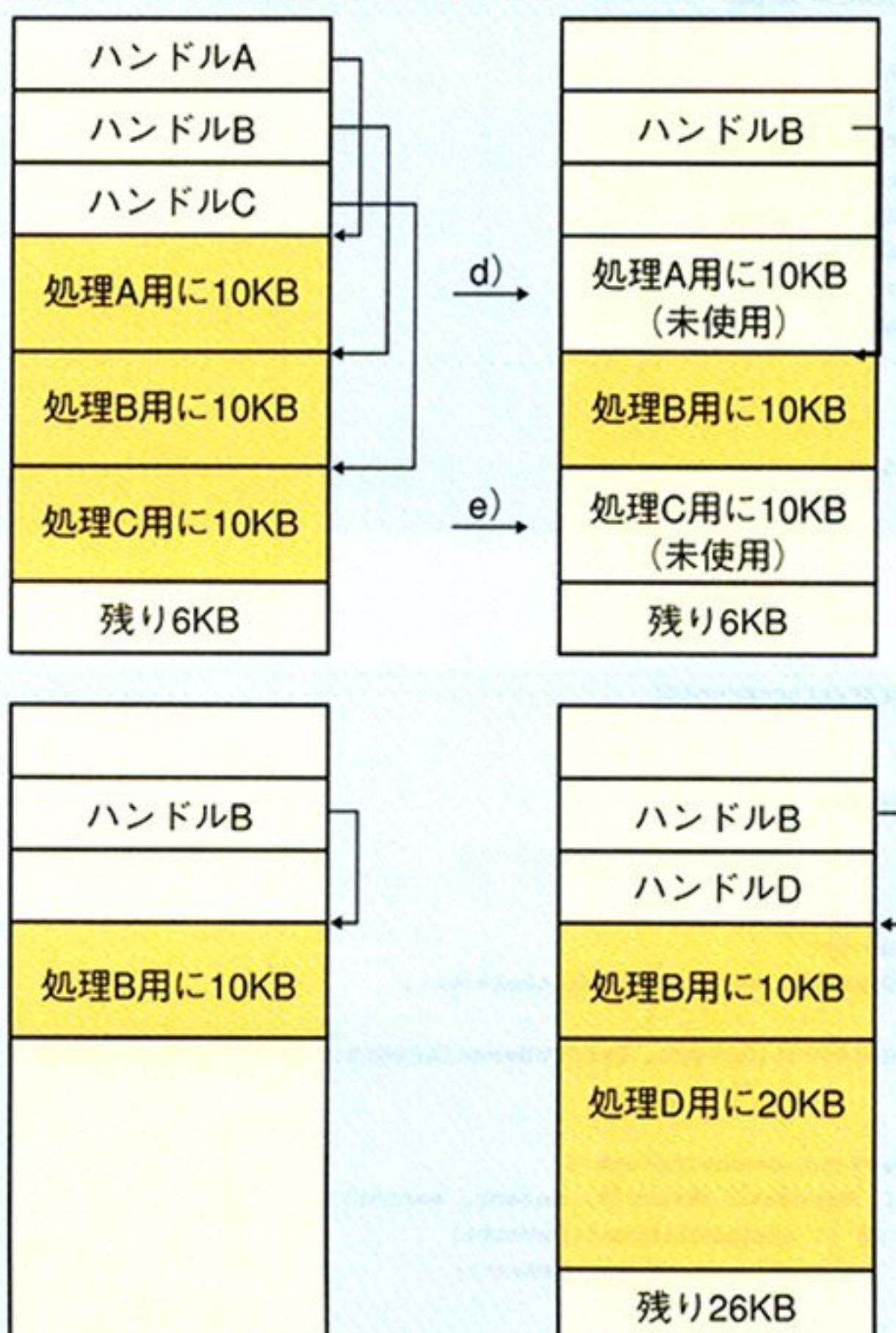
```
for (I = 0; I < 10; I++) {
    PutChar((＊hMemory) [I]);
}
```

と面倒でも，必ずメモリハンドルを用いたアクセスを行う(面倒であるばかりでなく，コードも大きくなりますし実行速度も遅くなります)か，メモリハンドルを使用している間中，

```
VoidPtr           pMemory = MemHandleLock( hMemory);
for (I = 0; I < 10; I++) {
    PutChar(pMemory[I]);
}
MemHandleUnlock(hMemory);
```

のように，MemHandleLock 関数でメモリの移動が発生しないようにします(「メモリハンドルをロックする」という)。ただし，不必要にロックを行うと十分にメモリコンパクションが行えなくなってしまうので，あるメモリハンドルをロックする必要がなくなったら，すみやかにMemHandleUnlock関数でロックを解かなければなりません。

※これは，SX-Window，Macintosh(やWindows 3.x)などのMMUがなかった頃に用いていた「メモリハンドル」の考え方と同じものです。

図2-1　メモリ領域をポインタでアクセスする場合

Application Iconの右にある「Create...」(あるいは「Edit...」)ボタンを押してください(図8)。

するとアイコン編集画面が表示されますので，図10のように，ひと目でGraffiti関連のプログラムであるとわかるようなアイコンへ変更します。

あとは，思う存分Graffitiの練習をしてください。ちなみに，私の最高得点は130点でした。

Graffiti練習プログラムを改良する

せっかく作ったGraffiti練習プログラムですが，あまりPalmアプリケーションっぽくありません。サンプルプログラムとはいえ，のっぺりとしていて単機能です。これは通常のPalmアプリケーションがメニューやボタンを用いて操作するのに対し，起動，即ゲームスタート，あとはキーボード入力だけという操作方法しかないのが理由でしょう。

「メニュー」を追加する

お手軽なところから，以下の「メニュー」を追加します。

- Options
  - Restart　　ゲームの再スタート
  - Config...　ゲームの設定
- Help
  - Graffiti Hint　　Graffitiのお手本
  - About...　「～について」のダイアログウィンドウを開く

図11がPalm OSでのメニュー部品の呼び名です。Windowsの場合，メニューバーとメニュー(項目)は明確に分離されておらず「Constructor」を使っている分には，どちらから操作を始めてもシームレスに作業が行え，あまり意識する必要が少ないですが，プログラミング時には念頭に置いていてください。特にフォームと関連づけるのは「メニュー」でなく「メニューバー」です。

先ほど，アプリケーションアイコンを変更したように，プロジェクトウィンドウの「Starter.rsrc」をダブルクリックし，「Constructor」を起動します。

リソース一覧が表示されるので，Menu Bars/Main Formをダブルクリックします(図12)。

すると，図13のようなメニュー編集画面が表示されます。

デフォルトでは，「Option → About Starter App」のメニューしかありません。ここで，先ほどのようにメニューバーとして「Option」のほかに「Help」を追加します。

「Constructor 」のメニュー(図9の細長いメニューバー)から，Edit → New Menuを選択します。すると「Untitled」と名づけられたメニューバーが追加されるので，名前を「Help」と変更してください。変更の方法は，エクスプローラを使ったファイル名の変更と同じで，変更したい文字列の上にマウスカーソルをしばらく置きます。すると，テキスト編集モードに入るので適当に編集を行えます。

次に，メニューバーにぶら下げるメニュー項目を作成します。

リスト3-3

```
void
GameStart(void)
{
   szTarget[0] = '\0';
   iLeft = 100;
   iMissed = iScore = 0;
   bGameOver = false;
}

void
GameOver(void)
{
   bGameOver = true;

   SndPlaySystemSound(sndError);

   FntSetFont(largeBoldFont);
   WinDrawChars("Game Over!", 10, 10, 110);
}

void
AddTargetChar(void)
{
   int c, i;

   if(iLeft > 0)
       iLeft--;
   else
       goto Over;

   c = SysRandom(0) % 26 + 'a';
   for(i = 0; i < sizeof(szTarget); i++) {
       if(szTarget[i] == '\0') {
           szTarget[i] = c;
           szTarget[i+1] = '\0';
           return; // Adding Done
       }
   }
Over:
   // Game over when there is no room to add the char.
   GameOver();
}

void
OnDraw()
{
   int     i;
   int     iSpaces = sizeof(szTarget) - StrLen(szTarget);
   char    szTmp[30];
   RectangleType  r = { 0, 70, 160, 20};

   FntSetFont(largeFont);

   StrPrintF(szTmp, "Left   : %3d  ", iLeft);
   WinDrawChars(szTmp, StrLen(szTmp), 10, 20);
   StrPrintF(szTmp, "Score : %3d   Miss : %3d", iScore, iMissed);
   WinDrawChars(szTmp, StrLen(szTmp), 10, 35);

   WinEraseRectangle(&r, 0);
   for(i = 0; szTarget[i]; i++) {
      WinDrawChars(&szTarget[i], 1, (i+iSpaces) * 10 + 10, 70);
   }
}

Boolean
CheckTargetChar(char c)
{
   int i;
   Boolean bHit = false;
   char    szTmp[20];
   int     iPoint = 1;

   // Blocks any inputs after "Game Over"
   if(bGameOver)
       return  false;

   FntSetFont(largeFont);

   StrPrintF(szTmp, "Input  (%c) ", c);
   WinDrawChars(szTmp, StrLen(szTmp), 10, 50);

ReSearch:
   for(i = 0; szTarget[i]; i++) {
       if(szTarget[i] == c) {
           strmove(&szTarget[i], &szTarget[i+1]);
           iScore += iPoint;
           iPoint ++;
           SndPlaySystemSound(sndClick);
           OnDraw();
           bHit = true;
           goto ReSearch;
       }
   }

   if(bHit == false) {
       // Mis typo!
       iMissed++;
       if(iLeft > 0) iLeft--;
   }
   return bHit;
}

void
strmove(char *p, char *q)
{
   do {
       *p++ = *q;
   } while(*q++);
}
```

Cold Fireとは？

私が日々SX-BASICのメンテナンスに追われていた頃，嬉々としてアセンブラでSX-Windowsのアプリケーションを書いていた人々がいました。メモリが2MBあったにも関わらずにです。

Palm OSでは，これよりもメモリが少ないですし，実行サイクルが少なくて済むということは電池が長く保つということですので，アセンブラプログラマの腕の見せどころです。特に，X68000で鍛えた68000系のアセンブラテクニック大技，小技を遺憾なく発揮できます(私は軟弱者なのでしばらくC言語は放せません)。

というわけで，Oh!Xの読者として気になるのは，Cold Fireと68000はどのくらい似ているのかということでしょう。

典型的なCISCの680x0プロセッサに対し，Cold Fireは，

シリコンの小型化(低価格化？)

ほとんどの命令を１クロックで実行

命令長は，2/4/6バイトのいずれか

というRISC志向の改造を加えたプロセッサです。そのため，除算命令のようなシリコン面積を喰う命令はすぱっと削除されています。Cコンパイラはまず使用しないと思われるBCD演算関連も削除されています。また，ローテートシフト命令のように１クロックで終わらない命令も削除されています(68000ってバレルシフタを積んでなかったのね)。link/ulink命令は，C言語でよく使うだろうということで，生き残っています(個人的には，DBcc命令を残していただいたほうがありがたかったのですが……)。

具体的には，以下がCold Fireで省略された命令です。

ABCD	10進数加算
AND to CCR	CCRとの即値論理積
ANDI to SR	SRとの即値論理積
CHK	レジスタ境界チェック
CMPM	メモリ内容どおしの比較
DBcc	デクリメント付き条件付き分岐
DIVS	符号付き除算
DIVU	符号なし除算
EORI to CCR	CCRとの即値排他的論理和
EORI to SR	SRとの即値排他的論理和
EXG	レジスタの交換
MOVE to CCR	CCRへの転送
MOVE SP	SRへの転送
MOVEP	ペリフェラルからのデータ転送
ORI to CCR	CCRとの即値論理和
ORI to SR	SRとの即値論理和
RESET	リセット
ROL	左シフト
ROR	右シフト
ROXL	左シフト(巡回)
ROXR	右シフト(巡回)
RTR	CCRの復帰付きリターン
SBCD	10進数減算
TAS	セマフォ更新
TRAPV	条件付きTRAP命令

逆に，Cold Fireで追加された命令も若干あるのですが，ほとんどが組み込みアプリケーションを作成するときに便利な命令ばかりで，Palmアプリケーションではほとんど使いません。

メニュー項目の作成は親となるメニューバー部分ををマウスで選択し，Edit → New Menu Itemで行えます。また，失敗しても，

Edit → Cut Menu Item　　で削除

Edit → Undo（事前の動作）でやり直し

が行えますので，各自で好きなように挑戦してください。文章で各手順を説明するよりも，実際にいろいろ試したほうが速くマスターできるはずです。

メニュー作成の順序は各自好きなように行えるのですが，名前は一字一句間違えないでください。というのは，アプリケーションプログラムで選択されたメニュー項目を判定するのに，リスト4-1のように，

「フォーム名＋メニューバー名＋メニュー項目名」

というラベルを使用するからです。このラベルはConstructorがメニュー項目の名前などから自動的に生成するので，それらの文字列を正しく入力しないとプログラムで使用されているラベル名と一致しなくなります。

メニューに関するデータを入力終えたところで，試しにビルドしてみましょう。「Hello World !!」のときと同様，＊.prcファイルをPOSEかPalmマシンに転送して実行します。

Palmでは，左上のアプリケーション名が書いてある領域をタップするか，左下の（液晶部分ではなく，Graffitiエリアの左）メニューの絵が描いてある部分をタップすメニューバーが現れます（図14）。

無事，表示されたでしょうか？　メニューを操作してもなにも起こらないはずです（あるいは，デフォルトのAboutダイアログが表示される場合もあるかもしれませんが，これはConstructorのバグ（？）ですので気にしません）。

「メニュー」の動作を追加する

メニューの表示ができるようになったことところで，それに対応する動作をプログラミングします。先ほどのスケルトンプログラムをよく見ると，

Boolean MainFormDoCommand (Word command)

関数でデフォルトのAbout画面を表示していますので，ここに各メニュー処理を追加するのがPalm OSプログラミングの流儀のようです。この関数は，

引数：選択されたメニュー項目のID

戻り値：メニュー処理を行った（True）か否か（False）＊5

を取ります。

先ほど，メニュー項目を４つ作成しましたが，まず，ゲームの再スタート

図12

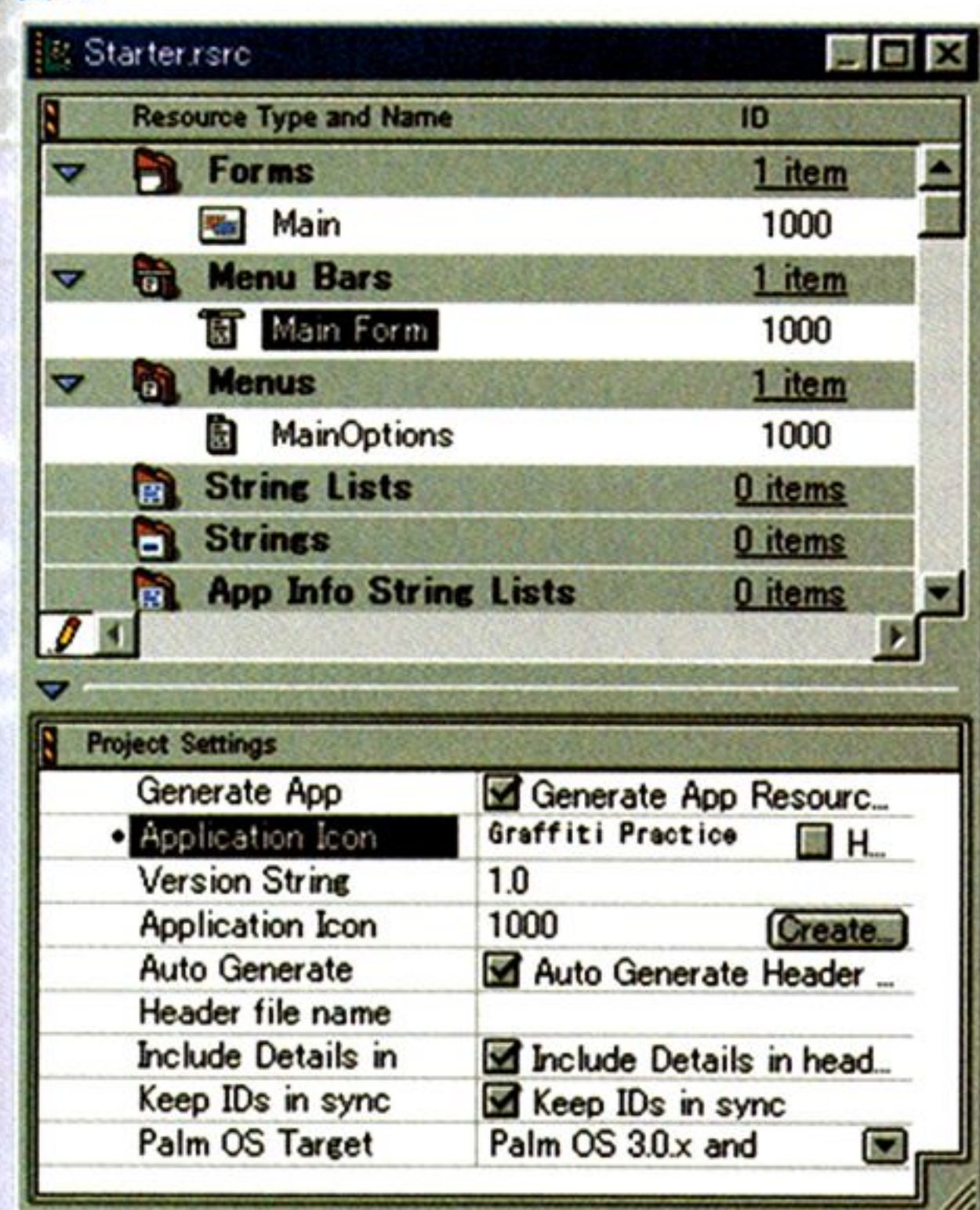
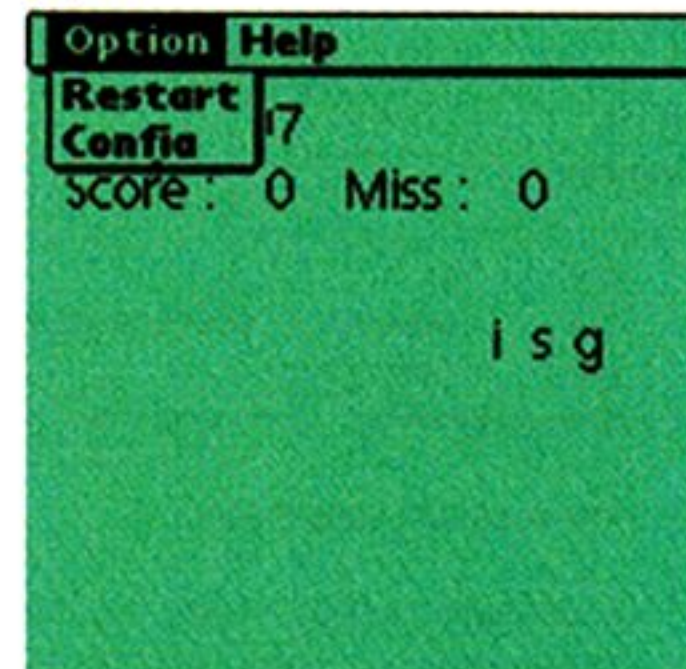



図13

図14

とGraffiti見本を表示する処理のみを追加し，残りの2つはあとでじっくり取り組みます。*6

この関数をリスト4-1のように変更し，再ビルドしようとすると，ラベルMainOptionsRestartとMainHelpGraffitiHintが見つからない，とエラーになるかもしれません。

どうやら，ConstructorとCode Warrior IDEとでリソースID定義ファイルの扱いが異なるようで，

Constructor →プロジェクトディレクトリ¥rsc¥Starter_res.h
Code Warrior IDE → プロジェクトディレクトリ¥src¥StarterRsc.h

と食い違っているようです。しかも，スケルトンプログラムと同時に，src¥StarterRsc.hが生成されるので，コンパイルエラーにもならずに，私も途方に暮れました。インストールのミスなのかなにが原因なのかはわかりませんが，同じ現象に出くわしたら，Starter.cの最初のほうの，

#include "StarterRsc.h"

を，

#include "..¥rsc¥Starter_Res.h"

と変更してください。私の個人的な予測ですが，StarterRsc.hは単に間違って生成されただけで，その後なにも使われていないようです。混乱を避けるために消去するか別ディレクトリに保存しておいたほうがよいでしょう。

それはさておき，

Option → Restart　でゲームの再スタート
Help → Graffiti Hint...　でGraffitiの見本一覧の表示

が正しく行われたでしょうか？

「About...」ダイアログの追加

追加するメニュー項目4項目のうち2つを追加しましたので，残りは，ゲーム設定ダイアログを開く，とAbout...ダイアログです。後者のほうが簡単なような気がしますので，こちらを先に実装しましょう。

スケルトンプログラムにあるように，AbtShowAbout()関数を使用すれば簡単に処理を行えるような気もするのですが，Palm OS SDKを見ると，この関数は「Warning! System Use Only」と赤太字で注意書きされていますので，潔くあきらめ，正攻法で実装します。

まずは，ダイアログウィンドウ（フォーム）のデザインです。Constructorのリソース管理ウィンドウ中，Formsをクリックし，Edit → New Form Resourceとメニュー選択し，新しいFormを作成します。Formの名前は「About Graffiti Practice」とします。

まっさらな画面いっぱいのウィンドウが作成されますが，About...ダイアログはそんなに大きい必要はないので，

Left Origin　2
Top Origin　70
Width　156
Height　88

と変更してください。

Palm OSの画面サイズが160×160ドットあるのに，Left Origin + WidthやTop Origin + Heightが158と2ドット分，少なく設定するのに気づかれたでしょうか？　ここで指定する値は，ダイアログウィンドウの枠線分を含まないため，Left Origin + Width＝160に設定すると，枠線がPalmマシンの画面をはみ出してしまい，表示されなくなってしまうからです。

画面サイズのほかにも，ウィンドウタイトルを「About...」としておきましょう（図15）。

次に，このダイアログに対し，「OK」ボタンや文字列を追加します。ConstructorのWindow → Catalogメニューで，「カタログ」ウィンドウを表示します（図16）。

ドロー系のお絵描きソフトでいうところの，ツールパレットと基本的に機能は同じなのですが，扱いがやや特殊です。アイテムを選んでウィンドウ上でレクタングルを描く，というMS-Windows系のアプリケーションとは異なります。正しいConstructorの使い方は，このカタログから適当なオブジェクトを選んでダイアログウィンドウ編集画面に「ドラッグ」します。

まずは，About...ダイアログですので，ダイアログを閉じるための「OK」ボタンを配置します。カタログウィンドウから，「Button」をドラッグし，ダイアログウィンドウの中央下部へ配置します。

細かい設定は，各自でアレンジしてかまわないのですが，私は，以下のように設定しました。

Object Identifier　OK
Button ID　1101
Left Origin　60

リスト4-1　メニュー処理ルーチン（その1）

```
static Boolean MainFormDoCommand(Word command)
{
   FormPtr frmP;
   Boolean handled = false;

   switch (command)
      {
#if 0
      case MainOptionsAboutStarterApp:
         MenuEraseStatus (0);
         AbtShowAbout (appFileCreator);
         handled = true;
         break;
#endif
      case MainOptionsRestart:
         GameStart();
         handled = true;
         break;
      case MainHelpGraffitiHint:
         SysGraffitiReferenceDialog( referenceAlpha );
         handled = true;
         break;
   }

   return handled;
}
```

図15

Form 1100, "About Graffiti Practice"

Layout Properties	
Form	
Left Origin	2
Top Origin	70
Width	156
Height	88
Usable	Usable
Modal	Modal
Save Behind	Save Behind
Form ID	1100
Help ID	0
Menu Bar ID	0
Default Button ID	0
Form Title	About...

Layout Appearance

図16

Catalog
UI Objects
UI Object Name
Button
Checkbox
Field
Form Bitmap
Gadget
Grafitti Shift Indicator
Label
List
Popup Trigger
Push Button
Repeating Button
Scrollbar
Selector Trigger
Table

Top Origin　　　70
Width　　　36
Height　　　12
Usable　　　☑ Usable
Anchor Left　　　☑ Anchor Left
Frame　　　☑ Frame
Non-bold Frame　　　☑ Non-bold Frame
Font　　　Standard
Label　　　OK

次に，プログラムの説明文を追加します。カタログウィンドウから，「Label」をドラッグし，ダイアログウィンドウの中心部分に置いてください。

私は以下のように設定しました。

Object Identifier　　　App Name
Label ID　　　1102（←システムが自動的に割り振るので，異なった値が得られても変更してはいけない）
Left Origin　　　50
Top Origin　　　30
Usable　　　☑ Usable
Font　　　Bold
Text　　　Graffiti Practice
　　　Version 1.0

寂しいので絵を加えます。カタログウィンドウから「Bit Map」をドラッグし，画面左中央に配置します（図17）。

Object ID　　　1103
Object Identifier　　　Unnamed1103
Left Origin　　　15
Top Origin　　　40
Bitmap Resource　　　1000
Usable　　　☑ Usable

これだけでは真っ白な空白ですので，Bitmap Resource 1000と表示されている右隣にあるCreate...（あるいはEdit..）ボタンを押して，表示する絵を作成します。先ほどのApplication Iconと同じものを設定しておけばよいでしょう。

ここで，ファイルをセーブし，Constructorを終了します。

このAbout...ダイアログは，メニューから開かれることになっていますので，リスト4-1のメニュー処理関数内に，リスト4-2のような処理を追加します。

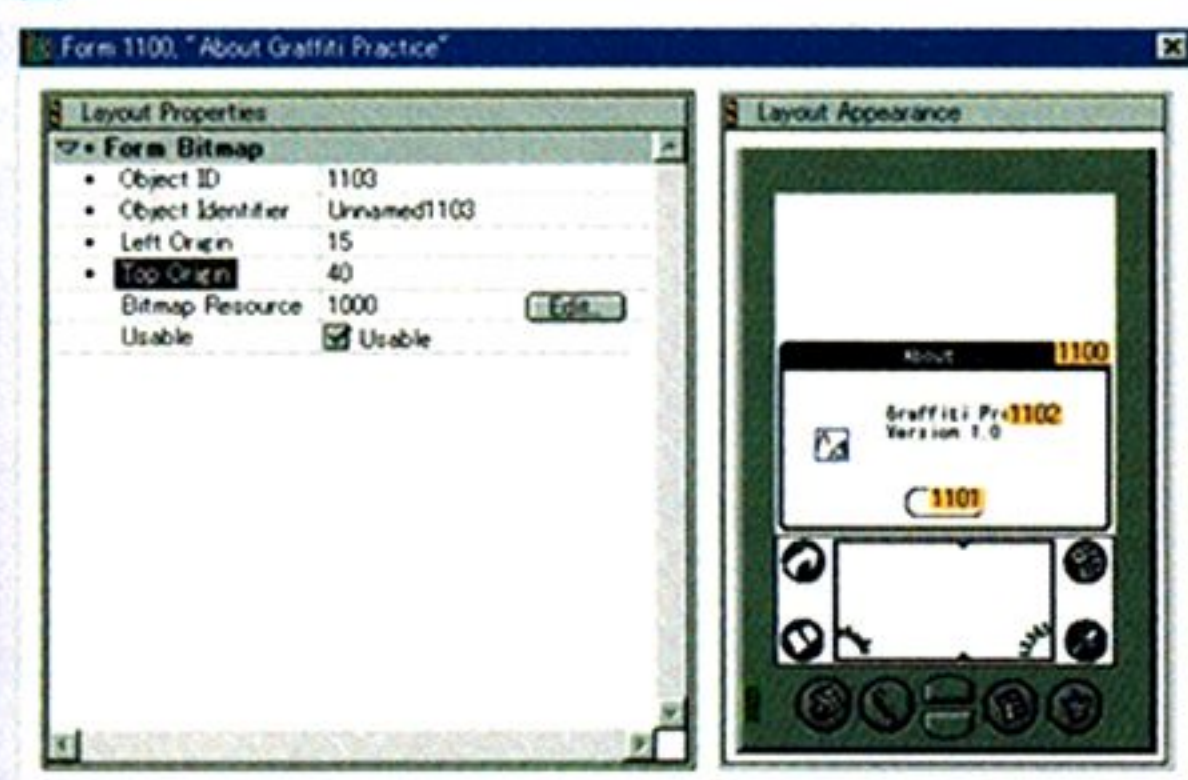
MenuEraseStatus (CurrentMenu);

About...ダイアログ表示中に，ゲームの再スタート行えたりすると，不自然ですし，About...ダイアログを２重に開かれても困りますので，メニュー表示を一時的に禁止します。

frmP = FrmInitForm (AboutGraffitiPracticeForm);

これで，フォームリソースからFormType構造体を作成します。AboutGraffitiPracticeFormというのは，Constructorが自動生成した定数でAbout...ダイアログのリソースIDが定義されています。

FrmDoDialog (frmP);

図17

指定したFormType構造体のフォームを表示します。これはWindowsでいうところのModal Windowですので，OKボタンが押されるまで関数は終了しません。

FrmDeleteForm (frmP);

About...ダイアログのFormType構造体をメモリ上から消去します。

さっそく，再ビルドして，実行してみましょう。

正しく，About...ダイアログが表示されていますね？（図18）

図18

Preference

Preferenceとは，日本語で好みとか嗜好という意味です。Palm OSでは，アプリケーションの設定などを意味します。Graffiti練習プログラムでは，文字列の出現スピード，文字列の種類（小文字のみ，大文字＋小文字，大文字＋小文字＋特殊記号）を選べるようにします。

このような設定は，一度設定したら，次回プログラム起動時にも残っているとよいですね。そのためには，これらの設定をアプリケーション終了時に保存，2回目以降起動時に読み込みを行います。

ファイル操作（Palm OSではデータベースと呼ぶ）で行えないこともないのですが，Palm OSにはこのような機能を実現するための機能がすでに備わっているので，こちらを使用します。ちょうどWindowsで設定を保存するのにファイルでなくレジストリを使用するのに似ています。このレジストリに相当するメカニズムをPalm OSではPreferenceと呼ぶのですが，スケルトンプログラムのなかにすでに雛形は作られています。

AppStart()関数のPrefGetAppPreference()を呼び出しているところが起動時にPreferenceの設定を読み出すことところです。また，AppStop()関数でPrefSetAppPreferences()を呼び出していることろが次回起動時のために設定を保存していることころです。

これらのアクセス関数は1項目ごとではなく，StarterPreferenceType構造体を用いて，すべての設定を一括して行うようになっています。

スケルトンプログラム中では，StarterPreferenceType構造体の変数がローカル変数として宣言されていて，アクセスごとにグローバル変数をコピ

リスト4-2　About...ダイアログ表示ルーチン

```
case MainHelpAboutGraffitiPractice:
   // Load the info form, then display it.
   frmP = FrmInitForm(AboutGraffitiPracticeForm);
   FrmDoDialog(frmP);

   // Delete the info form.
   FrmDeleteForm(frmP);
   handled = true;
   break;
```

リスト4-3　Preferenceの保存，読み出し

```
static Err AppStart(void)
{
    StarterPreferenceType prefs;
    Word prefsSize;


   // Read the saved preferences / saved-state information.
   prefsSize = sizeof(StarterPreferenceType);
   if  (PrefGetAppPreferences(appFileCreator, appPrefID, &prefs, &prefsSize,
true) !=
      noPreferenceFound)
      {
      MemMove(&thePrefs, &prefs, prefsSize);
      }

   return 0;
}

static void AppStop(void)
{
   // Write the saved preferences / saved-state information.  This data
   // will be backed up during a HotSync.

   PrefSetAppPreferences (appFileCreator, appPrefID, appPrefVersionNum,
      &thePrefs, sizeof (thePrefs), true);
}
```

ーしなさいという意図だと思われるのですが，今回の設定項目は文字のスピードと種類だけですから，StarterPreferenceType構造体自身をグローバル変数としてしまいましょう。

```
typedef enum tagGAME_MODE {
        MD_LOWERS_ONLY,
            MD_LOWERS_AND_UPPERS,
            MD_ALL
} GAME_MODE;

typedef enum tagGAME_SPEED {
        MD_SPEED_SLOW,
        MD_SPEED_NORMAL,
        MD_SPEED_FAST
}GAME_SPEED;

typedef struct
{
        GAME_MODE       m_iMode;
          GAME_SPEED    m_iSpeed;
} StarterPreferenceType;

StarterPreferenceType    thePrefs = {
          MD_LOWERS_ONLY,
          MD_SPEED_NORMAL
        } ;
```

これで，プログラム中どこでもPreference情報はthePrefs.m_iMode（文字の種類），thePrefes.m_iSpeed（スピード）としてアクセスすることが可能になりました。これらを保存，読み込みする関数をリスト4-3のように変更します。

コントロールを追加

Preference情報を保存する変数は確保できましたので，いよいよGUI的にPreferenceを変更できるようにします。About...ダイアログの作成したときと同じ要領でゲーム設定用のウィンドウを作成します（図19）。

```
Left Origin          2
Top Origin           30
Width                156
Height               128
Usable               [v] Usable
Modal                [v] Modal
Save Behind          [v] Save Behind
Form ID              1200
Help ID              0
Menu Bard ID         0
Default Button ID    0
Form Title           Game Configuration
```

Listアイテムは少々扱いに戸惑うかもしれませんが，図20のように設定してください。

　カタログウィンドウから，Listオブジェクトをダイアログウィンドウへドラッグする
　プロパティウィンドウ中のList Itemという項目をクリックしながら，Constructorのメニューバーから，Edit → New Item Textを選択

すると，アイテム項目が入力できるようになりますので，

```
Speed
     Item Text1     Fast
     Item Text2     Normal
```

図20

図19

図21

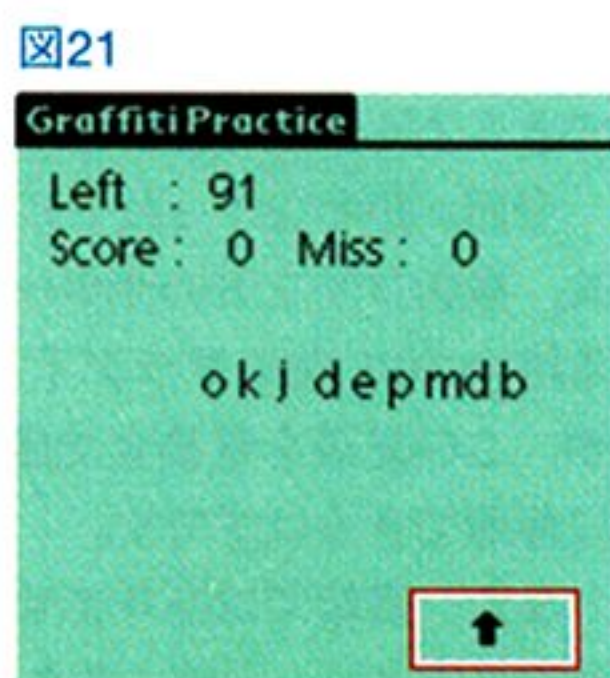
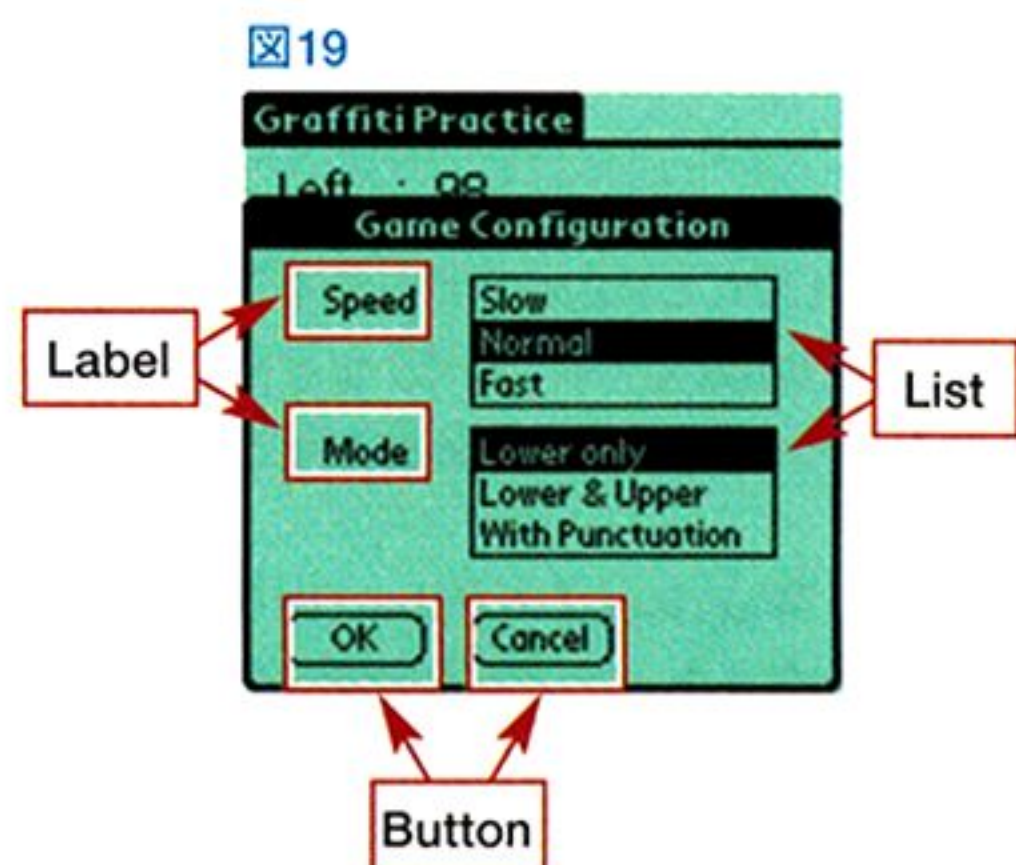
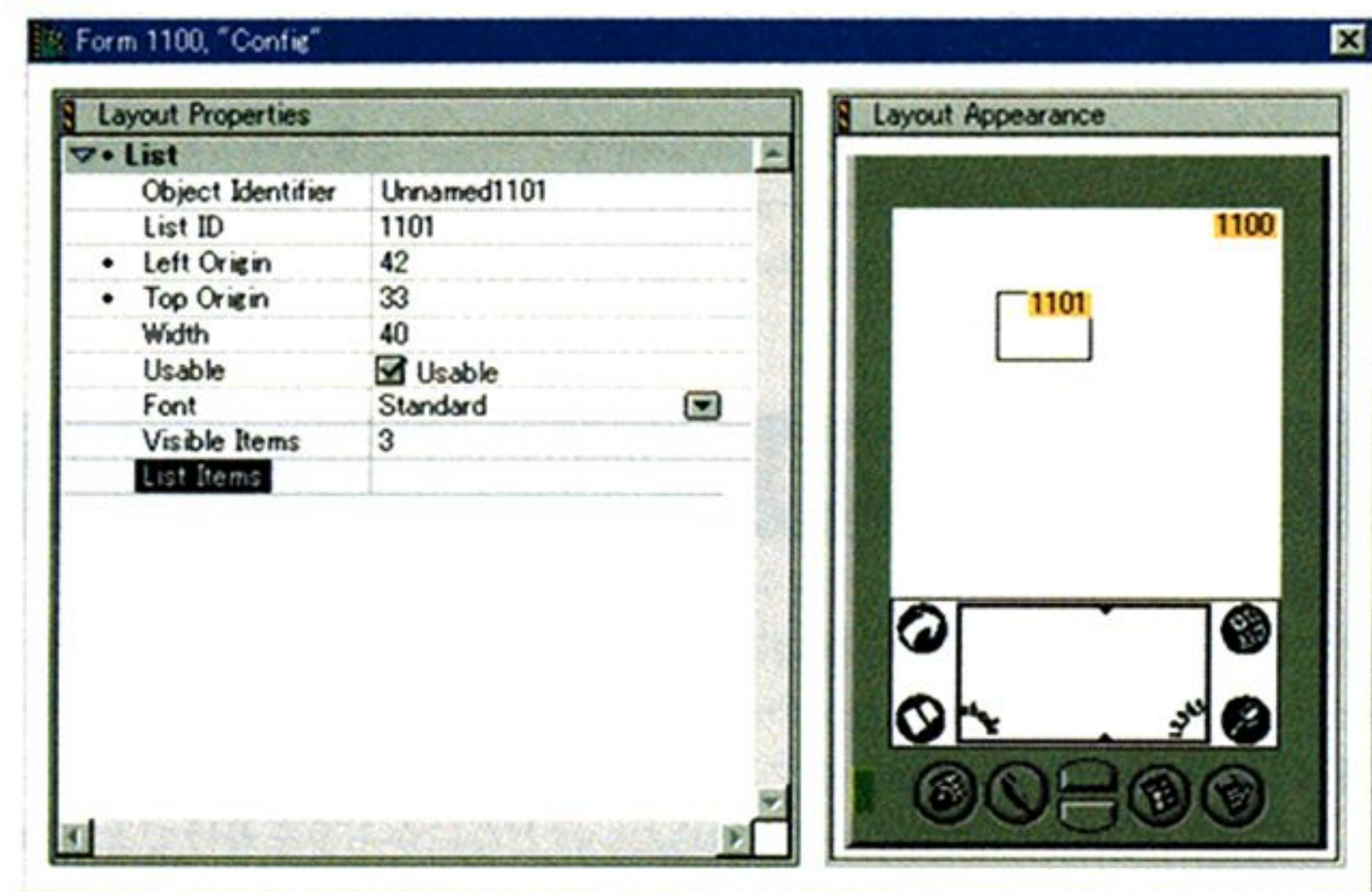



表5　設定ダイアログにあるオブジェクト各設定

Label	
Object Identifier	SpeedText
Label ID	1201
Left Origin	20
Top Origin	20
Usable	[v]Usable
Font	Standard
Text	Speed
Label	
Object Identifier	ModeTExt
Label ID	1202
Left Origin	20
Top Origin	60
Usable	v Usable
Font	Standard
Text	Mode
List	
Object Identifier	Mode
List ID	1203
Left Origin	60
Top Origin	60
Width	80
Usable	[v]Usable
Font	Standard
Visible Items	3
List Items	
Item Text1	Lower only
Item Text2	Lower & Upper
Item Text3	With Punctuation
List	
Object Identifier	Speed
List ID	1204
Left Origin	60
Top Origin	20
Width	80
Usable	[v]Usable
Font	Standard
Visible Items	3
List Items	
Item Text1	Slow
Item Text2	Normal
Item Text3	Fast
Button	
Object Identifier	OK
Button ID	1205
Left Origin	10
Top Origin	110
Width	36
Height	12
Usable	[v]Usable
Anchor Left	[v]Anchor Left
Frame	[v]Frame
Non-bold Frame	[v]Non-bold Frame
Font	Standard
Label	OK
Button	
Object Identifier	Cancel
Button ID	1206
Left Origin	60
Top Origin	110
Width	36
Height	12
Usable	[v]Usable
Anchor Left	[v]Anchor Left
Frame	[v]Frame
Non-bold Frame	[v]Non-bold Frame
Font	Standard
Label	Cancel

Item Text3　　Slow

Mode

Item Text1　　Lowers Only

Item Text2　　Lower & Upper

Item Text3　　With punctuation

と入力してください。各アイテムの設定に関しては表5にまとめておきましたので参照してください。この際，座標は適当でよいのですが，プログラム中のラベル名と一致させるためObject Identifierだけは必ず一致させておいてください。

設定ダイアログを開く

以上のようにして作成したダイアログウィンドウを実際に使えるようにプログラミングします。ほかの処理に比べて，長くなりそうですので，独立した関数を用意します。この関数をOption → Config...とメニュー操作されたときに呼び出すので，リスト4-1に以下のようなcase文を入れてください。

```
case MainOptionsConfig:
    OnOptionConfig();
    handled = true;
    break;
```

リスト4-4　設定ダイアログの処理

```
void OnOptionConfig(void)
{
   FormPtr pPreviousFrm = FrmGetActiveForm();
   FormPtr pFrm = FrmInitForm(ConfigForm);
   Word    iResult;

   ListPtr pSpeed    =    FrmGetObjectPtr(pFrm,    FrmGetObjectIndex(pFrm,
ConfigSpeedList));
   ListPtr pMode     =    FrmGetObjectPtr(pFrm,    FrmGetObjectIndex(pFrm,
ConfigModeList));

   FrmSetActiveForm(pFrm);

   LstSetSelection(pSpeed, thePrefs.m_iSpeed);
   LstSetSelection(pMode, thePrefs.m_iMode);

   iResult = FrmDoDialog(pFrm);

   if(iResult == ConfigOKButton) {
       thePrefs.m_iMode = (GAME_MODE)LstGetSelection(pMode);
       thePrefs.m_iSpeed = (GAME_SPEED)LstGetSelection(pSpeed);
   }
   if(pPreviousFrm)
       FrmSetActiveForm(pPreviousFrm);
   FrmDeleteForm(pFrm);
}
```

リスト4-5　thePrefs.iGameModeとthePrefs.iGameSpeedの値を反映させる。

```
void
AddTargetChar(void)
{
   int c, i;

   if(iLeft > 0)
       iLeft--;
   else
       goto Over;

   if(thePrefs.iMode == MD_LOWERS_ONLY) {
       c = SysRandom(0) % 26 + 'a';
   } else if(thePrefs.iMode == MD_LOWERS_AND_UPPERS) {
       c = SysRandom(0) % 52 + 'A';
       if(c > 'Z') c += 6;
   } else {
       c = SysRandom(0) % 94 + '!';
   }

   for(i = 0; i < sizeof(szTarget); i++) {
       if(szTarget[i] == '\0') {
           szTarget[i] = c;
           szTarget[i+1] = '\0';
           return; // Adding Done
       }
   }
Over:
   // Game over when there is no room to add the char.
   GameOver();
}
```

実際にダイアログを開くルーチンは，リスト4-4のようになります。大まかな流れは，

1)FrmInitForm()関数でフォームリソースから，FromType構造体を作成
2)FrmDoDialog()関数で，ダイアログを開く
3)FrmDeleteForm()関数で，使用したFormType構造体をメモリ上から消去

となります。

ただし，今回は，ダイアログウィンドウ上にListオブジェクトがありますので，

1)と2)の間で，現在の設定をListコントロールに反映（LstSetSelection()関数）

ダイアログが「OK」ボタンにより閉じられた場合，Listコントロールの値を，現在の設定に反映（LstGetSelection()関数）

という2つの操作が追加されています。

あと，半分おまじないみたいなものですが，設定変更中のアクティブフォームをゲーム画面から設定ダイアログへと変更しています（FrmSetActiveForm()関数）。

以上でGraffiti練習プログラムのPreferenceが保存/編集できるようになったはずです。この段階でプログラムを再ビルドして実行すると，

Preferenceダイアログを開くと，前回の設定が保存されていること
ほかのアプリケーションを実行したあとでも，Graffiti練習プログラムの設定が残っていること

が確認できます。

設定が変更できるようになったので，その設定をゲームにも反映させます（リスト4-5）。

また，先ほどのAppEventLoop()関数内でnilEventを発行するタイミングですが，以下のように変数の戻り値を使用するようにします。

```
if (bGameOver)
    EvtGetEvent (&event, evtWaitForever);
else
    EvtGetEvent (&event, GetCharacterInterval());
```

Graffitiでは，英大文字/小文字を同じストロークで入力します。PCのキーボードでいうところのCaps Lockモード（ペンを下から上に1文字に引くと入る）を切り替えながら区別するのですが，このモードもゲーム中表示してあったほうが，親切でしょう。「Constructor」からMain Formを選択し，右下のほうにGraffiti Shift Indicatorオブジェクトを入れてください。このオブジェクトはアプリケーションプログラムが明示的に操作しなくても自動的に入力モードを表示するので（図21），配置するだけであとはなにも必要ありません。

まとめと参考資料

駆け足でPalmアプリケーションの作り方を見てきましたが，いかがだったでしょうか？　この記事を機会に，我々がせっせとSX-Windowアプリケーションを作っていた頃のユーザー中心のコンピュータ文化みたいなものを思い出していただければ幸いです。

本稿は，Palmプログラミングを始めたいけどなにから手を着けてよいのかわからない，親切な文書がたくさんあるのはわかるけど，ありすぎて困ってしまう，英語が苦手という人を念頭に置いて書きました。本稿でひととおりのトピックは網羅したはずですので，簡単なプログラムなら自分で作成できるようになっているはずです。

今回，扱わなかったトピックに，テーブル（表）コントロール，データベース，検索コマンドのサポート，Conduit（PCとPalmのデータをリンクさせる場合のPC側のプラグインでJavaかVisual C++で作る）などがありますが，この記事を最後まで読まれた方なら必要に応じて参考文献をスムーズに参照できるはずです。

また，書籍ではありませんが，Code Warriors for Palm OSをインストールするとハードディスクに以下のようなドキュメントも展開されるはずですので参考にしてください。

Palm OS Programmer's Companion (Companion.pdf)

Palm OSプログラミングを始めるに当たっての知っておくべきことなど。前半部分は，本稿とだぶりますが，より詳しく説明されています。後半には，データベース，シリアル通信，赤外線通信，TCP/IPネットワークなどのトピックが扱われているので，これらを使用するPalm アプリケーションを作成する場合は該当する章が参考になるでしょう。

Code Warrior Constructor for Palm OS Platform (Constructors for Palm OS.pdf)

Palmアプリケーションのリソース作成を受け持つ「Constructor」のマニュアルです。Constructor自身は，マニュアルを隅々まで読まなくても使えるようにGUI的に上手に設計されているのですが，ときどきWindowsユーザーにとって「！？」な操作を要求してくることもあります（たいていは，Macintoshユーザーにとっては自然らしいのですが……）。そのようなとき，必要に応じて参照すればよいでしょう。

Debugging Palm OS Applications on Windows (Palm OS Debugging on Win.pdf)

今回は話を簡単にするため，コンパイルしたプログラムは通常のプログラムと同じようにPalmマシンあるいは，エミュレータに転送して実行していましたが，実はCode Warrior for Palm OSには，たいへん強力なデバッガ機能があります。作ったプログラムが複雑になりすぎて，内部変数の値を1つひとつ確かめたいときなど，デバッガの助けが必要なときに参照するとよいでしょう。

Code Warrior Palm OS Tutorial Manual (Palm OS Tutorial Windows.pdf)

とても丁寧なチュートリアルです。メモ帳の各機能を1章ごとに丁寧に追加していきます。各章ごとにできているべきソースなども，別ディレクトリにコピーされていますので，n章のコードをコピーしてくれば，1～n-1章を飛ばして直接n章を試す，ということも可能です。1章から丁寧に読んでいってもよいですし，「メニューを追加するには？」「フォームにスクロールを追加するには？」と知りたいトピックを必要に応じて試すのもよいでしょう。

Palm OS SDK Reference (Reference.pdf)

いわゆるSDK（Software Development Kit）のReferenceです。Palm OSに用いられる起動コード，パラメータ引数，API関数などが網羅されています。語学学習でいうところの単語辞書みたいなものなので，最初から読破しようとせずに必要に応じて参照してください（1054ページあります）。

と，本当に付録CD-ROMに収録させてもらってよいのかと思うような親切かつ丁寧な文書なのですが，書かれているのが英語です。Logo Vista E to Jに少し翻訳させてみたのですが，あと一歩のところで意味がわかりません（往年の「Inside Macintosh（日本語版）」と同程度かややましかな……）。ところが，嬉しいことに日経BP社から「 Palm OS Programmer's Companion 」と「 Palm OS SDK Reference 」をまとめて翻訳した「Palm OSバイブル」という本が出版されていますので，英語のドキュメントが苦手な人は参照しましょう。また，日本におけるPalm OSの第一人者といわれる山田達司氏による監修ですので翻訳品質もたいへん高くなっています。

参考文献

1) Liz O'hara, John Schettiono「Palm OS Programming for dummies」IDG Book 1999

お父さんのためのWindows98入門やWord入門をシリーズ化しているなと思ったら，ここまできたかという感じで，Palm OSも刊行されました。Palmの使い方ではなくプログラミングの本です。オヤジギャグなのか言葉の問題なのか理解に苦しむ箇所が何カ所かありますが，構成はしっかりしています。サンプルプログラムは，ほとんどCode WarriorsとGccの両方で動くようになっています。数章を通じて個人名とPID（Personal ID）データベース（アドレス帳のようなもの）を作っていますので，データベース関連のアプリケーションを作成するときには大いに参考になるでしょう。

付録のCD-ROMには，サンプルプログラムのほかに，Code Warriors Lite，関連GNUツールが収録されています。特に，GNUツールではクリック一発で環境整備が行え，おまけにTutorialまでついたパッケージが収録されています。ここに収められたgccのバージョンは，Palm OS用にチューンされたものらしく，各関数でa6レジスタと保存するためにコールバック宣言のマクロとうんちゃらかんちゃらという操作が不必要で便利です。また，インターネットで，自分でこれらのツールを別々に集めようとすると，まず，Windows上でUNIXライクな環境を構築してBashというシェルを走らせて……と面倒なのですが，これらの作業が本書の付属CD-ROMではひとつにパッケージされています。

2) 漆畑　広樹「ここまでできるPalm / WorkPadプログラミング Windows版」オーム社

日本語で読める軽めの入門書です。本稿は理解した，しかしSDKのドキュメントは難しすぎる，と感じた人はこれを読むとよいでしょう。参考文献1と同様「ここは日本だ日本語化」「戦えCode Warrior」「Constructorは建設省」など意味不明な言葉が多く見受けられますが，図を多く使いCode Warriorを用いたプログラム開発の各手順をを丁寧に説明しています。サンプルプログラムでは，インクエディタ（メモを文字でなく，筆跡そのまま残す。Windowsのペイントをメモ代わりに使うイメージ）を作成しています。守備範囲は本稿＋データベースといった感じです。

3) Neil Rhodes, Julie Mckeeban著，青木　龍也監訳，佐藤　信彦訳「Palm プログラミング　Palm / WorkPadアプリケーション開発ガイド」オライリージャパン（製作：日本ルーセントテクノロジ，発売元：オーム社）1999

全500ページというボリュームです。Conduit（PalmとPCをリンクさせるときに必要なPC側のプログラム）まで含めて，Palmプログラム開発に必要な情報はすべて含まれるといってよいでしょう。ただし，初心者向けとはいい難く，ひととおりPalmに関してマスターしてからチャレンジするとよいでしょう。あるいは，ある特定のジャンルの詳しい説明が必要なときに該当する章を開くという使い方も役立つでしょう。

オライリーの翻訳本というと，シリーズの動物の表紙を思い出すとともに，日本語訳に不安を抱く方がいるかもしれませんが，本書に関しては日本語の品質に問題はまったくありません。ちなみに表紙は「カワラバト」です。

メニューを出すタイミング

Palm OSのSDKなどを読んでいると，「MenuHandleEvent()関数がmenuEventイベントを発行する。アプリケーションはそのメンバ変数 ->menuIDで，どのメニューが選択されたか調べること」みたいな記述がありますが，これを整理すると以下のようになります。

A.　Palm OSではメニューは勝手に処理されない

B.　メニュー処理はMenuHandleEvent()関数が呼ばれたときに行われる（スケルトンプログラムのイベントループ関数AppEventLoop()で，確かに呼んでいますね）

C.　MenuHandleEvent()関数は選択されたメニュー項目を戻り値として返さない。ただし，メニューが実際選択された場合（プルダウンメニューからでも，ショートカットキーからでも），「メニューが選択された」というイベント（menuEvent）をイベントキューに足す

D.　アプリケーションはイベントループ中で，C.で作成されたイベントが回ってきたら対応する処理を行う。

Palm OSでは，各メニュー項目が固有のID（コマンドID）を持っているのですが，その値がイベントとしてメッセージキューに入れられます。

このようにメニュー項目に固有の値を割り当ててイベントとして処理すると，(1) メニューとして選択された場合，(2)ショートカットキー入力で選択された場合の両者を同じイベントとして検出でき，共通のサブルーチンで処理することが可能です。

5）　基本クラスに追加の処理関数を足していけるMCFと違い，親関数はこの値を見て，
　　ユーザー関数でなんらかの処理が行われたので，デフォルト処理を行わない
　　ユーザー関数でなんら処理が行われなかったので，デフォルト処理を行う
の切り分けを行います。

6) Graffitiの見本ダイアログも専用にダイアログを開くウィンドウですので処理は難しいはずなのですが，実際はSysGraffitiReferenceDialog()というシステム関数があり，この関数を呼び出すだけですべての処理をPalm OSが行ってくれます。やはり，多くの人がGraffitiで苦闘しているのでしょうね。

ケータイエージェント Jumonとは？

菊地 功 Kikuchi Isawo

Javaを搭載したさまざまな機器のあいだで通信に関するいろんな面倒なことをまとめてやってくれるミドルウェア，それがJumonです。エージェントという概念がどんなものかについても見ていきましょう。

昨年末から携帯電話でJavaが動くようになったらしいというのは，もう一方の記事でも述べた。それを見据えてか，オムロンソフトウェア株式会社から，Javaの通信環境ミドルウェアが発表された。正式名称はAgent Based Distributed Middleware ケータイエージェントJumonという。

名称といい，イメージキャラクターといい，実にナニな感じだが，そのナニな雰囲気とは裏腹に，実際には結構使えるんじゃないかというのがその筋の見解だ。現在はバージョン0.9の評価版が無料で配布されており，本誌CD-ROMにも収録できたのでちょっとだけ紹介しよう。

Javaというとネットワーク言語という印象が強く，実際にそういった部分を得意としている。が，その多機能性と，さらにC言語ライクな本格言語仕様も手伝って，そういったネットワークプログラミングは容易ではない。たとえば，Javaに限らずリモート通信を行うプログラムを作ろうと思ったら，ソケットの生成や，オブジェクトの送受信など，さまざまな手順が必要となり，その際にデータやスレッドの管理といった面倒な手間が増え，アプリケーション依存の使い回しの利かないものになってしまう（とドキュメントには書いてある）。実際，ネットワークゲーム特集のときにはJava担当の霧雨氏が地獄を見たらしいので（氏はそれなりのJavaのプロである），説得力はある。

そういった部分を一手に引き受けてくれるのが，ORB（Object Request Broker）というミドルウェアの概念だ。ORBを用いれば，通信部分を部品化できるため，既存のアプリケーションにリモート通信機能をつけたり，より汎用性の高いアプリケーションを容易に作れるようになる。

また，家電にもネットワークが浸透してくれば，複数の機器を順に制御したいといったことも出てくるだろう（別に家電でなくてもいいのだが）。たとえば，衛星チューナのチャンネルをあわせ，ビデオの録画ボタンを押し，セレクタをあわせ，テレビを点けるといった場合など。それぞれの端末に対してサーバからいちいちリモート通信を行うことも可能だが，あまり効率的ではない。また，電子レンジで1分温める，といった時間のかかる処理の場合はサーバがその間待たされることになる。

こういったことを効率よく処理する方法としてモバイルエージェントという概念がある。これは特定の環境で動いていたプログラム（オブジェクト）が処理の途中でネットワークを介して別の環境に移動し，続きの処理を行うというものだ（図1）。「プログラムが移動？」というとピンとこないかもしれないが，実際にプログラムを見てもらうと意味がわかってくるだろう。

ところで，別の端末に移動して続きを行うというには，同一のプラットフォームが必要ということだ。仮にネイティブのプログラムでやろうと思ったら，レジスタやスタックまでも移動前の状態を復元しなくてはならなくなる。つまり，ハードウェア，ソフトウェア（OS）がまったく同一でなくてはならない。そこでJavaの登場というわけだ。仮想マシンであるJavaVMならば，JavaVMのレイヤーによってそういった差異を吸収してくれる。まさにうってつけだ。

こういったORBやモバイルエージェントの概念を実現するJavaのミドルウェアがJumonなのだ。ミドルウェアといういい方が曖昧ならば，もっとぶっちゃけてクラスライブラリといってもいい。Jumonを使えば，TCP/IPがどうだとか，スレッドがこうだとか，（あんまり）意識しないで済む。なお，ここでいうJavaとは，Javaアプレットのことではなく，Javaアプリケーションのことである。ま，Javaアプレットもアプリケーションの一種だから使えないことはないと思うけど。

図1　モバイルエージェントの動作の概念

機能

ひと口にJumonといってもいくつかのパッケージが存在する。ベースでありコアであるJumon，それにいくつかの拡張機能が付加されたJumon-Turbo，携帯電話などの小型端末で機能するJumon-Miniである。さらに，Jumon-Miniにはその全機能を包含するL_JumonMiniと，サーバへのリモートメソッドで代替するS_JumonMiniがある。ここでは基本であるJumonについて簡単に説明しよう。とはいえ，実際にJumonが持つ機能はそれほど多くはない。主なものは以下の5つだ。

1.リモートオブジェクト生成

クライアント側から，サーバ側に（Jumon自体はクライアントサーバになれるので，要するに接続相手に）特定のオブジェクトの生成を要求する。ここでいうオブジェクトとはクラスのインスタンスを表す。

2.リモートメソッドコール

1で生成したリモートオブジェクト，あるいはサーバ側で生成され，次で説明するネーミングサービスを行ったオブジェクトのメソッドをコールできる。このメソッドの処理は，もちろんサーバ側で処理されるので，サーバを制御したり，サーバ側の情報を収集してリターンできる。

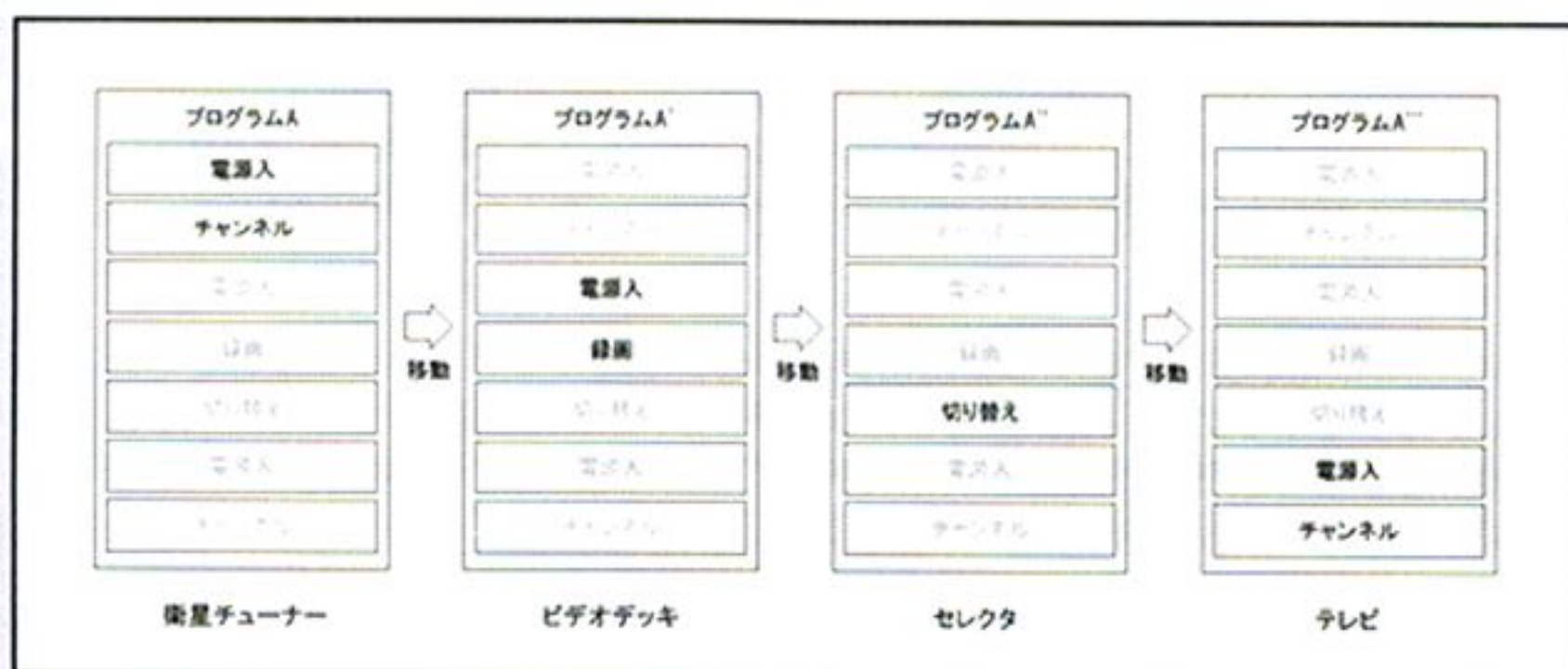
3.ネーミングサービス

生成したオブジェクトに対して任意の名前をつけておくことで，外からその名前でオブジェクトを参照できる。

4.モビリティ

クライアント側で作成したオブジェクトをサーバに移動させる。移動後はリモートメソッドコールで制御できる。

5.エージェント

オブジェクト自身にメソッドとして移動経路やアクションを組み込み，自律的に端末を移動して制御を行う。

エージェントは特殊としても，それ以外はサーバ側を意識させせない分散処理のための機能だ。こういったことを自前でやろうとすると，それはそれは大変なことになるが，Jumonを使えば恐ろしく簡単にできてしまう。順番に重要なメソッドを説明しよう。

```
static void Jumon.startup ();
static void Jumon.startup (
```

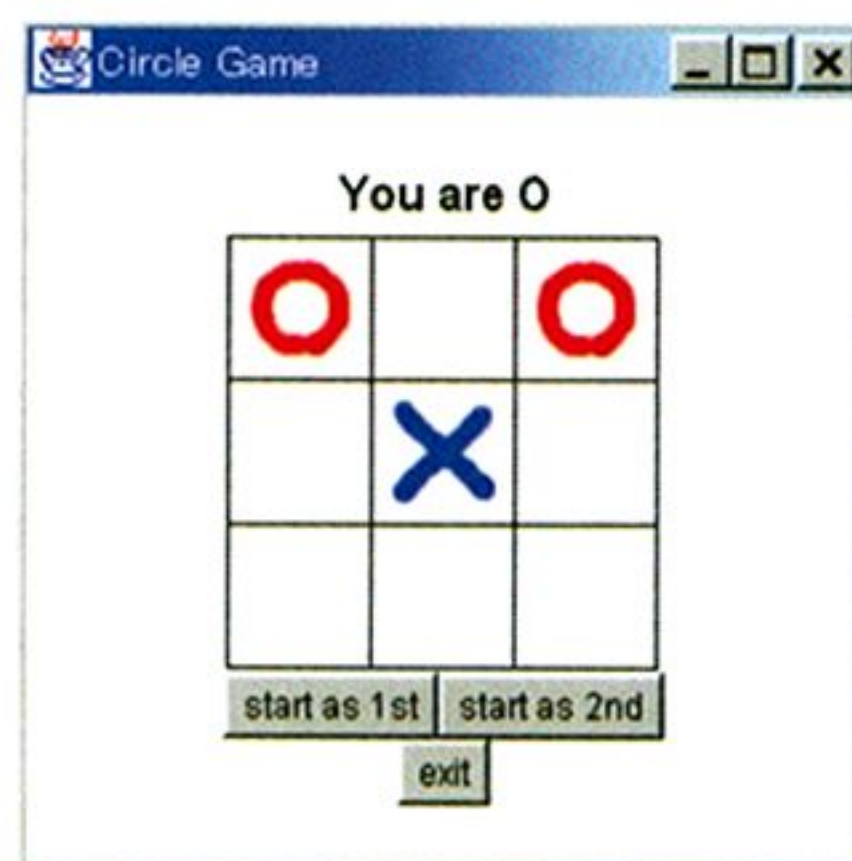



図2　Jumon付属のサンプル○×ゲーム

String url);

Jumonの初期化であり，Jumonを使う場合にはまず最初に必要である。引数を省略するとクライアント専用（送信はできるが受信はできない）となり，urlを指定するとクライアントサーバとなる。一般にはポート番号を指定するようだ。リモート制御のポート番号を7000の場合は次のようになる。

Jumon.startup ("7000");

static Proxy Factory.create (String classname, String url);

urlで指定したサーバ上にclassnameで指定したクラスのインスタンスを生成し，そのプロキシを返す。プロキシとは，リモートのオブジェクトを遠隔操作するための，いわばラジコンの送信機のようなものであり，classnameで指定したクラスのインタフェイスが返る。このインタフェイスに対して，普通にメソッドをコールするだけでリモートメソッドをコールできる。たとえばThunderbirdというリモートマシンのポート7000を通してSecondクラスのインスタンスを生成し，DropContainerメソッドをコールするには次のようにする（ISecondはSecondクラスのインタフェイス）。

```
ISecond is = (ISecond)Factory.create ( "Second", "//Thunderbird:
7000" );
is.DropContainer ();
```

なお，プロキシオブジェクトはダイナミックプロキシ機能により動的に生成されるが，その場合はリモートオブジェクト生成時に反応が悪くなるので，コンパイル後にあらかじめpgenコマンドでプロキシクラスを生成しておくのがベター。

static void Namespace.bind (String name, Object object);

objectで指定したオブジェクトに，リモートで参照できる名前をつける。クライアント側からはリモートオブジェクトとして操作されるので，オブジェクトはlocalhost上にFactory.create ()で生成しておく。たとえば，Jumonをポート番号7000で初期化したサーバ上に，Secondクラスのオブジェクトを作り，クライアントからはVirgilとして参照できるようにするには，以下のようにする。

```
ISecond is = (ISecond)Factory.create ( "Second", "//localhost:7000" );
Namespace.bind ( "Virgil", is );
```

static Object void Namespace.lookup (String name);

サーバ側にネーミングサービスでnameとして登録されたオブジェクトをクライアント側から検索し，そのプロキシを返す。上の例で挙げたオブジェクトを取得するなら（サーバ名はThunderbird），以下のようになる。

```
ISecond is = (ISecond)Namespace.lookup ( "//Thunderbird:
7000/Virgil" );
```

static IMobility Mobility.of (Object object);
void IMobility.MoveTo (String url);

objectで指定したオブジェクトのMobilityインタフェイスを取得し，urlで指定した先に移動させる。localhostで生成したオブジェクトをポート番号7000を開いたSD-Xに移動させるなら，

```
ISecond is = (ISecond)Factory.create ( "Second", "//localhost:7000" );
IMobility mobility = Mobility.of ( is );
mobility.MoveTo ( "//SD-X:7000" );
```

となる。移動後もプロキシisを使い，

```
is.resecue ();
```

などと（移動元で）するだけで，移動先で処理を行うことができる。

なお，実際にはオブジェクト（インスタンス）がすべてネットワークを介して転送されるわけではなく，同じクラスのインスタンスが転送先で生成され，クラス変数などの状態のみが転送，インスタンスが自動的に転送前の状態に初期化される（移動元のオブジェクトは，移動後に解放される）。そのため，移動するクラス（例ではSecondクラス）は，Serializableインタフェイスをインプリメントしておかなければならない（のだと思う）。

static IAgent Agent.of (Object object);
void IAgent.MoveTo (String url, String callback);

Objectで指定したオブジェクトのAgentインタフェイスを取得し，urlで指定した先に移動したあと，（転送先で）callbackメソッドをコールする。callbackメソッド内でさらに自分自身のAgentインタフェイスを取得し，移動させることで自律して移動させることができる。そのため，これまでの例でいえば，Secondクラス内で，次のようにするのが一般的。

```
IAgent agent = Agent.of ( this );
agent.MoveTo ( "//SD-X:7000", "resecueSD_X" );
```

resecueSD_Xメソッド内で必要な処理を行い，最後に今度は，

```
IAgent agent = Agent.of ( this );
agent.MoveTo ( "//Melchior:7000", "resecueMelchior" );
```

すれば，続いてMelchiorに移動しresecueMelchiorメソッドの処理を継続して行える。Serializableインタフェイスについては，Mobilityと同様。

この辺りについてはJumonに簡単なサンプルが付属しており，ドキュメントで解説されているので，実際にそれを動かしてみるといいだろう。筆者もなにかサンプルを作りたかったのだが，いいアイデアが浮かばなかったのと，時間がないので見送った。っていうか，○×ゲームを作ろうと思ったんだけど，サンプルに入ってたんだよね。3人以上接続したら観戦モードとか，わりかしちゃんと作ってあるようだ。結構参考になるんじゃないかと思う。なお，コンパイルには別途Java2SDKが必要なので注意。

標準ミドルウェアになりうるか？

このように，Jumonを用いればネットワークゲームや分散処理を，ほとんどそれを意識することなく記述できる。httpにも対応しているので，サーバを立ててダウンロード実行したり，ゲームロビーも作れる。一度接続してしまえば，あとはサーバを介さずに直接ユーザー同士で通信できるのでサーバの負荷も軽減でき，レスポンスを犠牲にすることもないだろう。

また，携帯電話に関しては，Javaとはいってもサブセットだったり，実はTCP/IPではなかったりということもあるようだが，その辺もうまい具合に吸収してくれそうだ。今年あたりからは，このJumonがあらかじめ携帯電話に組み込まれそうだという噂もある。本格通信プログラマからすれば，全然機能が足らないというのもあるのかもしれないが，少なくともアマチュア日曜プログラマにとっては十分楽しめそうだ。

ちょっと不安があるとしたら，セキュリティ関係かな。ウイルスがエージェントとして勝手に動き回って，勝手にクラッキングしまくってくれたら，ちょっとシャレにならん。その辺考えてないわけじゃないとは思うけど多少不安は残る。話を聞くとJumonの開発ではそのあたりでもっとも苦労したみたいなのだが（発表会でもわざわざウイルスとの違いとか説明してたし），世の中，万全と思っていてもなにがあるかわからないし。まあそれだけいろんなことができるということで，可能性が広がるのは確かなんだけど。

なお，Jumonは商用とかでなければある程度自由に使ってもかまわないようだ。ネットワークゲームを手軽に作ってみたいという人や，多機種間のネットワーク接続でなにかやりたいと考えている人は触っておくのも悪くないだろう。携帯電話などを含めると，今後のネットワークでの基本インフラとなる可能性もかなり高いので，基本的な概念くらいは把握しておきたい。

WindowsCEのCPU

川相直人　Kawai Naoto

携帯機器のCPUについてはあまり公表されることもない。最初から複数のCPUサポートを前提としたWindowsCEというのはきわめて異例な存在といえるだろう。WindowsCE系列機にはさまざまなCPUによる実装が行われ，組み込み系CPUの激戦区ともなっている。ここではWindowsCEで使われているさまざまなCPUを通して，携帯系機器に要求されているCPUのあり方を見てみよう。

はじめに

本号は携帯機器の特集ということで，WindowsCEに採用されているCPUについて解説する。携帯機器は機能が第一であり，それに採用されているCPUがスポットを浴びることは少ない。動作周波数が話題になる程度であろうか。WindowsCEの市場自体はそれほど大きいものではない。しかし，その市場に参入しようとしているCPUはかなり多く，生存競争は熾烈である。そのうち自然淘汰されていくと思われるが，現状は各メーカーの(x86系を除く)高性能CPUが鎬を削っている最中である。このいちばん面白い時期に存在しているCPUの特徴を知っていても損にはならない(得にもならないが)。なお，本稿はCPUのみに注目しており，それを搭載したWindowsCEマシンの優劣を比較しているのではないことを付け加えておく。また，本文中に「噂」という語句を何度か使っているが，それらの根拠は本当に薄弱なので，真剣には受け取らないように(だったら書くな，といわれそうだが)。

本稿は1999年の終わりに執筆されている。その後，順次加筆・修正を行ってはいるが，一部(かなり？)の情報に古いものがあることをお断りしておく。本稿は2000年5月11日以降は修正していないので，それを念頭に置いて読んでほしい。その後の最新情報は脚注で対応している。ただし，「おわりに」のみは8月24日に執筆した。

WindowsCEとは

WindowsCEとはマイクロソフトが開発した，携帯用の小型PC(ハンドヘルドPC，H/PC)やPDA(パームサイズPC，PsPC)，あるいは家電，ゲーム機向けのOSである。CEとは発表当時は「Consumer Electronics(家電)」の略とされていたが，現在のマイクロソフトの公式コメントでは「CEとはひとつの概念を表すものではなく，〝Compact(小型)，Connectable(ネットワーク接続)，Compatible(Windowsとの互換性)，Companion(Windowsとの連携)〟といった多くの設計思想を含む」とある。

表1　WindowsCEのバージョン一覧

発表年	コードネーム	バージョン	説　明
1996	Pegasus	1.0	H/PC。画面サイズ480×240ドット，モノクロ
1997	Alder	1.0	開発者用
	Birch	2.0	開発者用
	Mercury	2.0	H/PC。256色，またはモノクロ16階調表示 画面サイズは最大640×240ドット
1998	Gryphon	2.0	PsPC 1.1。初めてのパームサイズ版
	Dragon	2.0?	Dreamcast用
	Birch SP1	2.1	開発者用。CE2.11の原型？
	Orion	2.11	PsPC。詳細不明
	Jupiter	2.11	H/PC Pro。VGA，SVGAに対応。65,536色表示 電子メールにファイルの添付機能
	Apollo		カーナビ用。PCI，USBに対応
	Callisto		H/PCの原型？　詳細不明
	Callisto·		Jupiterを基にしたOS?
1999	Wyvern	2.11	PsPC 1.2。PsPC1.1のカラー対応版
	Venus	2.11	TVセットトップボックス用
	Hermes	2.11	次世代Web電話用
2000	Brich SP2	2.12	詳細不明。組み込み用途？
	Rapier	3.0	PsPC1.3？　デスクトップライクな機能を提供 ユーザーインタフェイスの改善 Pocket Officeを標準装備。高速動作
	Cedar	3.0?	H/PC。リアルタイム機能のサポート UPnP(Universal Plug and Play)に対応 CEF(CE Executive Format)をサポート
	Galileo	3.0?	H/PC。詳細不明。Cedarの置き換え？

さて，WindowsCEは，Windows95/98/NTとは別ものであるが，それらと同じ技術を使って設計されており，プリエンプティブなマルチタスクで動作する。Windows95/98/NTのプログラムは動作しないが，扱うデータには互換性がある(制限があるが)。発表当時は，Windows95/98/NTのデータをそのまま持ち運べるという点で注目された。HDDを使用せず，OSやアプリケーションはROMに格納されているため，起動が高速であり，電池寿命も長い，という特徴がある。

1996年の秋に発表されて以来，WindowsCEは幾度かのバージョンアップがなされてきた。現行のバージョンはCE2.11である。WindowsCEのバージョンとは主にカーネルのバージョンを指し，同じバージョンでも，ゲーム機用，H/PC用，PsPC用，AutoPC用といったハードウェアごとに異なる機能を有している。表1にWindowsCEのバージョンと特徴を示す。コードネームは幻獣の名前が多いが，惑星，衛星の名前もちらほら見受けられる。ペガサス，ドラゴンがあるので，ガリレオの次あたりにきっとタイタ

表2　WindowsCEがサポートしているCPU

Vendor	CE 2.0	CE 2.1	CE 2.11	CE 2.12	CE 3.0
AMD	Am486DX5 K6 ElanSC400 ElanSC410	Am486DX5 K6E ElanSC400 ElanSC410	Am486DX5 K6-2 ElanSC400 ElanSC410		
ARM		ARM720T	ARM720T	ARM720T ARM720T (Thumb)	ARM720T
Cyrix		Media GX MMX Media GX	MII		
Hitachi	SH3	SH3 SH4	SH3 SH4	SH3 SH4	SH3 SH4
Link-Up			7201	L7200	
IBM			PPC403GC	PPC403GC	PPC403GC
Intel	Pentium MMX Pentium DX4 486 SX	MMX Pentium SA1100	MMX Pentium SA1100	MMX Pentium SA1100	Pentium MMX Pentium Pentium II SA1100
Motorola	MPC821	MPC821 MPC823 MPC850 MPC860 PPC823a PPC860MH	MPC821 MPC823e MPC860 MPC860T PPC823a PPC860MH	MPC821 MPC823 MCP823e MPC850 MPC860 MPC860T	MPC821
IDT		RC64474 RC64475			
NEC	VR4100 VR4101 VR4102 VR4111 VR4200 VR4121 VR4300	VR4102 VR4111 VR4300 VR4310	VR4102 VR4111 VR4121 VR4181 VR4300 VR4310	VR4102 VR4111 R4111 (MIPS16) VR4121 VR4121 (MIPS16) VR4300 VR4310	VR4111 (MIPS16) VR4300
Philips	Poseidon v1.0 Poseidon v1.5	Poseidon v1.0 Poseidon v1.5	Poseidon v1.0 Poseidon v1.5	Poseidon v1.0 Poseidon v1.5	
QED		RM5230	RM5230 RM5231 RM5261 RM5271	RM5230 RM5231 RM5261 RM5271	
ST Micro		STPC Client	STPC Client		
Toshiba	TX3912	TX3912 TX3922	TX3912 TX3922	TX3912 TX3922 TX4955	TX3912
最終更新	02.15.2000	03.30.2000	02.22.2000	04.18.2000	05.05.2000

http://www.microsoft.com/windows/embedded/ce/guide/processors/default.asp による。

ンもあるのかな(わかる人にしかわからない文章だな)。

●プリエンプティブなマルチタスク：タイマ割り込みにより，複数のプログラムの実行を切り替えていくマルチタスク方式。非協調的マルチタスクともいう。反意語であるノンプリエンプティブなマルチタスクとはアプリケーション自身が制御を管理し，都合のいいときに次のプログラムに制御を切り替える。SX-Windowでプログラムを作ったことのある人はそれを思い浮かべればよい。それに反し，プリエンプティブなマルチタスクでは，ハードウェア(CPU)が強制的にプログラムの制御を切り替えるので，アプリケーションがマルチタスクであることを意識する必要はなく，プログラムの構造が簡単になる。WindowsNTやWindows95/98，UNIXなど，ほとんどのOSで採用されているマルチタスク方式はプリエンプティブである。

●WindowsCEの最新バージョン：2000年6月15日，マイクロソフトはWindowsCE3.0の出荷開始を発表した。リアルタイム機能およびマルチメディア機能の向上，対応言語の追加，「コンポーネント化」の強化が図られている。コンポーネント化とは，OSを細分化し，開発者が必要な個所のみを利用できるようにする仕組みだそうだ。現状では『Pocket PC』と命名されたPsPCで採用されている。これは『Rapier』というコードネームで呼ばれていたものである。同時期に発表されたH/PC用のOSである『Cedar』の消息は不明。

WindowsCEに使われるCPU

マイクロソフトはWindowsCEに使用するCPUを「認定」という形で許可している。これらはWindowsCEのバージョンごとに認定されている。マイクロソフトが認めていないCEのバージョンとCPUの組み合わせでは動作の保証もなく，製品として発売することもできない。それでは，どのようなCPUが認定されているのか。

それは，マイクロソフトのWindowsCEのサイトで知ることができる。表2にマイクロソフトが認定しているCPUの一覧を示す。また，それぞれのCPUの特徴を表3に示す。表3のデータは各メーカーのプレスリリースやデータシートの値を示している。情報源は主にインターネットであるが，日本のメーカーのサイトで値を公表してない場合でも米国のサイトでは公表している場合もあり，データシートをダウンロードしまくって表を埋めていった。

表3　各CPUの特徴

[86系]

名前	周波数	性能	電源電圧	消費電力
STPC Client	66MHz/75MHz	公表せず	3.3V	3.2W (66MHz)
ElanSC400 ElanSC410	33/66/100MHz	公表せず	3.3V	703mW (33MHz)
Am486DX5	66/100/133MHz	公表せず	3.3V	152mW (66MHz)
K6-2E	233/266/300/333MHz	公表せず	2.2V	8.75W (266MHz)
K6E	233/266MHz	公表せず	2.2V	8.75W (266MHz)
MII	300/333/366/400/433MHz	公表せず	2.9V	?

[MIPS系]

名前	周波数	性能	電源電圧	消費電力
VR4100	DC〜40MHz	45MIPS@40MHz	2.2〜3.3V	120mW (Normal) ? (Standby)
VR4101	33MHz	40MIPS@66MHz	3.3V	250mW (Normal) ? (Standby)
VR4102	66MHz	80MIPS@66MHz	3.3V	250mW (Normal) ? (Standby)
VR4111	80MHz/100MHz	130MIPS@100MHz	2.5V (内部I/F) 3.3V (外部I/F)	180mW (Normal) ? (Standby)
VR4121	131MHz/168MHz	210MIPS@168MHz	2.5V (内部I/F) 3.3V (外部I/F)	350mW (Normal) ? (Standby)
VR4122	180MHz	216MIPS@180MHz	1.8V (内部I/F) 3.3V (外部I/F)	270mW (Normal) ? (Standby)
VR4300	100MHz	125MIPS@100MHz	3.3V	1.8W
VR4310	167MHz	222MIPS@167MHz	3.3V	2.3W
TX3912	75MHz/92MHz	98MIPS@92MHz 80MIPS@75MHz	3.3V	350mW (Normal) 50 μW (Standby)
TX3922	129MHz/148MHz	192MIPS@148MHz 167MIPS@129MHz	2.7V (内部I/F) 3.3V (外部I/F)	440mW (Normal) 100 μW (Standby)
RM5230	175MHz	233MIPS@175MHz	3.3V (内部I/F) 3.3V (外部I/F)	4W
RM5231	250MHz	325MIPS@250MHz	2.5V (内部I/F) 3.3V (外部I/F)	2.7W (200MHz)
RM5261	266MHz	345MIPS@266MHz	2.5V (内部I/F) 3.3V (外部I/F)	3.6W (200MHz)
RM5271	266MHz	345MIPS@266MHz	2.5V (内部I/F) 3.3V (外部I/F)	4.2W (200MHz)
Poseidon (PR31500)	40MHz	公表せず	3.3V	363mW (Normal) 33 μW (Sleep)
Poseidon (PR31700)	75MHz	公表せず	3.3V	363mW (Normal) 33 μW (Sleep)

表からわかるように，CPUはx86系，MIPS系，ARM系，SH系，PowerPC系の5種類に大別できる。x86系以外は，いわゆる，メジャーな組み込み用RISCのCPUばかりである。x86自体は，いうまでもなく超メジャーである。コンシューマ向けマシンのCPUとしてクローズアップされることは少ないが，稼働している装置はそこそこあると聞いている。PowerPC系はサポートされているのは知っているが，こちらは実際のマシン上で稼働しているかは不明である。

x86系の事情は知らないが，WindowsCEの開発当初から，日立とNECが，それぞれSH3とVR4100で製品化に取り組んでいる。特に，日立は非常に熱心で積極的に共同開発を進めたため，現在でも，CPUの認定用の評価ボードはSH3用を基本にした設計になっているという。実際，WindowsCEの最初の製品は，SH3を採用したカシオのCASSIOPEIAシリーズと，VR4100系を採用したNECのMobilePro(日本ではMoibleGear)シリーズというH/PCである。それ以外のStrongARMやSH4といったCPUの製品が登場するのはWindowsCE2.1以降である。また，WindowsCEマシンがポピュラーになったものCE2.0以降である。それぞれのCPUの特徴を簡単に述べておこう。

●MIPS系

本誌の読者にはあえて説明の必要もないが，MIPS社が設計したRISCアーキテクチャを有するCPUである。MIPSというとスタンフォード大学での研究成果に裏付けされた高性能CPUというイメージがある。R3000やその64ビット版であるR4000は高性能EWSに搭載され一躍有名になった。その後，MIPS系のCPUは組み込み制御分野への方向転換を図り，周辺機能を内蔵した派生品が数多く登場する。日本ではNECと東芝が積極的に製品開発を行っていた。

現在販売されているCEマシンに搭載されているCPUは，NEC製のVR41xx系プロセッサと東芝製のTX39xx系のプロセッサである。海外用ではPhilipsのPoseidon(R3000系)もあったが，PhilipsはWindowsCEから撤退してしまった。このほかにMIPS系のCPUとしてはQED製のRM52xx系もある。これはWebTVやセットトップボックス用として想定されているが，搭載製品の販売はまだされていない(と思う)。少なくとも日本では。

マイクロソフトはMIPS系のCPUは，最低MIPS IIの命令セットアーキテクチャを要求している。VR41xx系はMIPS IIIのR4000系の流れを汲んでいるので問題ない。TX39xxは母体がMIPS IのR3000なのだが，Branch Likey命令がサポートされ，MIPS II相当になっている。また，インタロッ

[ARM系]

名前	周波数	性能	電源電圧	消費電力
ARM710T ARM720T	50MHz	45MIPS@50MHz	3.3V	180mW
ARM740T	40MHz	36MIPS@40MHz	3.3V	130mW
SA1100	133MHz/190MHz	220MIPS@190MHz	1.5V (内部I/F) 3.3V (外部I/F)	330mW (Normal) 75 μW (Sleep)
SA1110	133MHz/206MHz	235MIPS@206MHz	1.8V (内部I/F) 3.3V (外部I/F)	400mW (Normal) 88 μW (Sleep)

[SH系]

名前	周波数	性能	電源電圧	消費電力
SH3	100MHz	130MIPS@100MHz	3.3V	400mW (Normal) 250mW (Sleep)
SH3	133MHz	173MIPS@133MHz	1.9V (内部I/F) 3.3V (外部I/F)	400mW (Normal) 48mW (Sleep)
SH3	60MHz	78MIPS@60MHz	3.3V	400mW (Normal) 250mW (Sleep)
SH4	167MHz/200MHz	360MIPS@200MHz	1.8V (内部I/F) 3.3V (外部I/F)	2.8W (Normal) 500mW (Sleep)
SH4	128MHz	230MIPS@128MHz	1.5V (内部I/F) 3.3V (外部I/F)	400mW (Normal) ? (Sleep)
SH5	400MHz	604MIPS@400MHz	1.5V (内部I/F) ? (外部I/F)	600mW (Normal) ? (Sleep)

[PowerPC系]

名前	周波数	性能	電源電圧	消費電力
MPC821 MPC823	DC〜50MHz	66MIPS@50MHz	3.3V	?
MPC850 MPC860	DC〜66MHz	87MIPS@66MHz	3.3V	?
PPC403GC	25/33/40MHz	56MIPS@40MHz	3.3V	320mW

ク機能も備えているという。

●MIPS系のCPUを搭載したWindowsCEマシンである，シャープのTelios，JVCのInterlink，富士通のINTERTOPは，ブランドイメージを考慮してか，CPU名が公表されていない。しかし，本稿では前2つは東芝のTX3922，残りはNECのVR4121と断定して話を進める。マスコミに公表されたCPUのメーカー名や動作周波数を考慮すると，だいたい察しがつく。万が一間違っている場合はご容赦願いたい。INTERTOPに関してはNECの米国の関連会社であるNEC Electronicsのサイト(http://www.necel.com/)で暴露(？)されているので間違いない。ここには日本では入手できないドキュメントが数多くあり，VRファン(?)には注目のサイトである。

● SH3/SH4系

SH3とSH4は日立製のCPUである。従来のRISCは32ビットの命令長であったが，コード効率を重視した16ビット固定長の命令セットを採用する。命令セットは68000系との類似点が多く，アセンブラでのプログラム開発も容易である。その意味でマニアのファンが多い。SH自体は高性能と消費電力のバランスのよさをウリにしている。SHシリーズの最初のSH1は，TRONチップ用のFPU設計を担当した技術者が，オリジナルな高性能RISCを提供する目的で開発したと聞いている。その際，HennessyとPattersonの『コンピュータアーキテクチャ』が参考にされたという。その後，SH2では演算器の高速化とキャッシュが内蔵され，さらにSH3でMMUが内蔵されて，アーキテクチャは一応の完成を見る。SH4はSH3をスーパースカラ化してさらなる性能向上を達成した。なお，SH3とSH4はともにSH系と分類されるが，特権命令を除いても一部命令セット(というかアドレシングモード)に互換性がなく，(ユーザーモードにおいて)バイナリレベルで互換性がない。そのため，SH3のアプリケーションソフトは，SH4では，まず，動作しない(再コンパイルが必要)。このため，SH4を搭載したCEマシンであるPERSONAのユーザーはソフトウェアの少なさを嘆いている。さすがに最近では徐々にソフトウェアが増えてきているようではあるが。ところで，SH4の供給不足か，それほどの性能は不要と思われているのか知らないが，SVGAやVGAクラスのマシン以外ではSH4を見かけない。依然としてSH3が多く採用されている。やはり，プログラムの互換性を考えるとSH4は使いにくいのであろうか。あるいは戦略的な意図があるのかもしれない。

● StrongARM系

このプロセッサはARM社が設計した命令セットを，Alphaプロセッサで有名なDEC社がStrongARMとして実装し，現在はインテルが製造，販売を行っている。ARM自身は商業用に開発された最初のRISCである。基本アーキテクチャはカリフォルニア大学バークレイ校のRISC(SPARCの母体)を参考にしつつもCSICのよさを残したものになっている。条件コードを持ち，全命令の実行結果が条件コードに反映されるのが最大の特徴である。

現在のWindowsCEマシンに搭載されているのはSA1100というCPUで，190MHzという高速動作の割に330mWという低消費電力が特徴である。消費電力に重点を置くあまり，性能は二の次となっている感がある。ベンチマークテストによる性能は惨憺たるものだった。一方，Rapierに搭載予定なのはSA1110というCPUである。これはSA1100の高速版(206MHz動作)である。バスサイクルの周波数が103MHzと高速なのでそれなりの性能が期待できる。実際，「信じられないほどの高性能」という噂だ。

ところで，次期StrongARMはインテルが設計を行う。600MHz動作で450mWという超低消費電力を目標としている。インテルがCE分野に対してついに本気になったということであろう。ただ，StrongARMを採用している装置メーカーはHPしかなく，ソフトウェアの数が圧倒的に少ないのが欠点である。そのHPも最新機種ではSH3を採用している。ただし，最近では，この次期StrongARMに期待して，いまのうちからStrongARMに乗り換えようとしているCEマシンのメーカーも(かなり)あるようである。

CPUに求められる機能

WindowsCE用のCPUに求められる機能とはなんだろうか。現在，CE搭載マシンの主流はH/PCおよびPsPCであり，それはこれらのモバイルマシンに求められる機能であるともいえる。

● 高性能

いまに始まったことではないが，マイクロソフト製のOSは非常に重い。快適な操作環境を実現するためには相当なCPUパワーが必要である。試しに，手元のWindows98マシン(Pentium II 400MHz)のDhrystone MIPSを測定したところ，約300MIPS(gcc使用)だった。表3を見ればわかるが，WindowsCEのCPUは，周波数当たりの性能(つまり同一周波数で比較した場合)では，Pentium IIをかなり上回っている。Mobile Pentiumでも消費電力が6W以上(これは推定値)あり，Pentiumの動作周波数ではPDAには向かない。これらを考慮すれば，こんなに高性能を要求するOSはPDA用としてはWindowsCEのほかにはない。PDAを評するとき，「OSがCEでないのでサクサク動く」というのがほめ言葉になっているほどだ。ちなみに，Palm PilotのCPUであるDragonball(MC68328)は16.67MHz動作で2.7MIPSの性能である。WindoesCEのCPUと比べて100倍近い性能差があるのに体感性能はトントンかそれ以上。まあ，PalmOSはモノクロ4階調という利点(？)を差し引いても，OSの重さの違いは想像を超えているようだ。

WindowsCEに関して，なにをもって高性能というかは難しいところだ。最近ではCPUのMIPS値よりも，実際の体感性能，つまり，アプリケーションの起動時間や画面の切り替え時間が重要視されつつある。このとき，重要になるのはバスの転送速度(バンド幅)とキャッシュのヒット率である。

CPUメーカーとしては，MIPS値を上げるために，動作周波数を向上させたりスーパースカラの導入といった，アーキテクチャの改善に注力するより，単純にキャッシュ容量を増大するほうが遥かに効果的という，技術的には面白くない状況にあるようだ。キャッシュの構成をダイレクトマップから2ウェイセットアソシアティブに変更するのも効果的であることがわかっている。その割にはダイレクトマップ構成のキャッシュの採用が多いのは消費電力を考えてのことである。

CEは重いという批判に対応してか，マイクロソフトでは次期PsPC用OSである「Rapier」では大幅な性能改善を計画しているという(すでに過去形になってしまった)。OS自体のチューニングでは限界があるのか，噂では付属するPocketOfficeのアプリケーションにも手を入れて体感性能を向上させているとか(この情報は未確認)。ただし，サードパーティのアプリケーションやフリーソフトなどは考慮されていないのが問題である。Rapierは今年のWinter CES 2000でJornada430se版が公開された。触った人の感想では速いのか遅いのかよくわからないということだった(ということは，ずば抜けて高速というわけではないようだ)。なお，RapierではPalmOSに対抗して使いやすさも追求されているという。マイクロソフトのeBook電子ブックリーダーやMP3オーディオプレイヤーなどのマルチメディアサポートも充実しているらしい。Rapierの出荷は4月または5月の予定なので本誌が発売される頃には全容を現しているはずなのだが(追記3を参照のこと)。

● MMUの内蔵

WindowsCEは，Windows95/98/NTと同じく，プリエンプティブなマルチタスクを実現している。このためにMMUが必須である。SH3などはWindowsCE市場に参入するためにわざわざMMUを内蔵した製品を開発したと表明している。とはいえ，WindowsCEでは，一度に表示されるウィンドウはひとつのみ，しかも最大サイズで表示される。また，サイズの変更はできない。これでは一度にひとつのアプリケーションしか動かすことができず，マルチタスクである意味があまりない。組み込み制御分野全体がWindowsCEの応用分野なので一概には断定できないが，H/PCやPsPCではMMUというのは性能を低下させる要素でしかないと思う(TLBミスの処理が意外に重い)。CEが重く，サクサクと動作しない理由のひとつはおそらくここにある。なお，DragonballにはMMUなんて面倒臭いものは内蔵されていない(68000のメモリ保護機能は使用されていると思うが)。

NetNewsの投稿による未確認情報によるとWindowsCEは，タスクごとに仮想アドレス空間が定義されるという，多重仮想空間構造を採用していないらしい。信憑度50%程度だが，もし本当なら(仮想アドレス空間がひとつしかないなら)MMUを採用する意味はほとんどない。いくらマイクロソ

フトでもそんな無駄なことはしないと思うが，MMUなしで済ませられるならば，そうしてほしいものである。

●低消費電力

H/PCにしろPsPCにしろ，携帯を目的としているので，電池駆動が前提である。そこで電池寿命が製品を語るうえでの重要な要因になっている。いくら性能がよくても消費電力が大きいと使いものにならない。そこで，CPUの性能指標として単位電力当たりのMIPS値(MIPS/W)が使われることがある。図1に日立がWebサイト(`http://www.super-h.com/virtual_expo/seminar/ess1/01_1.html`)に掲載しているMIPS/Wの一覧を示す。表3の値と矛盾する部分もあるが日立の(宣伝用)資料なので，あまり深く考えないように。傾向はあっていると思われる。図中でPentiumIIの値については筆者が追加したMobile版の推定値である。

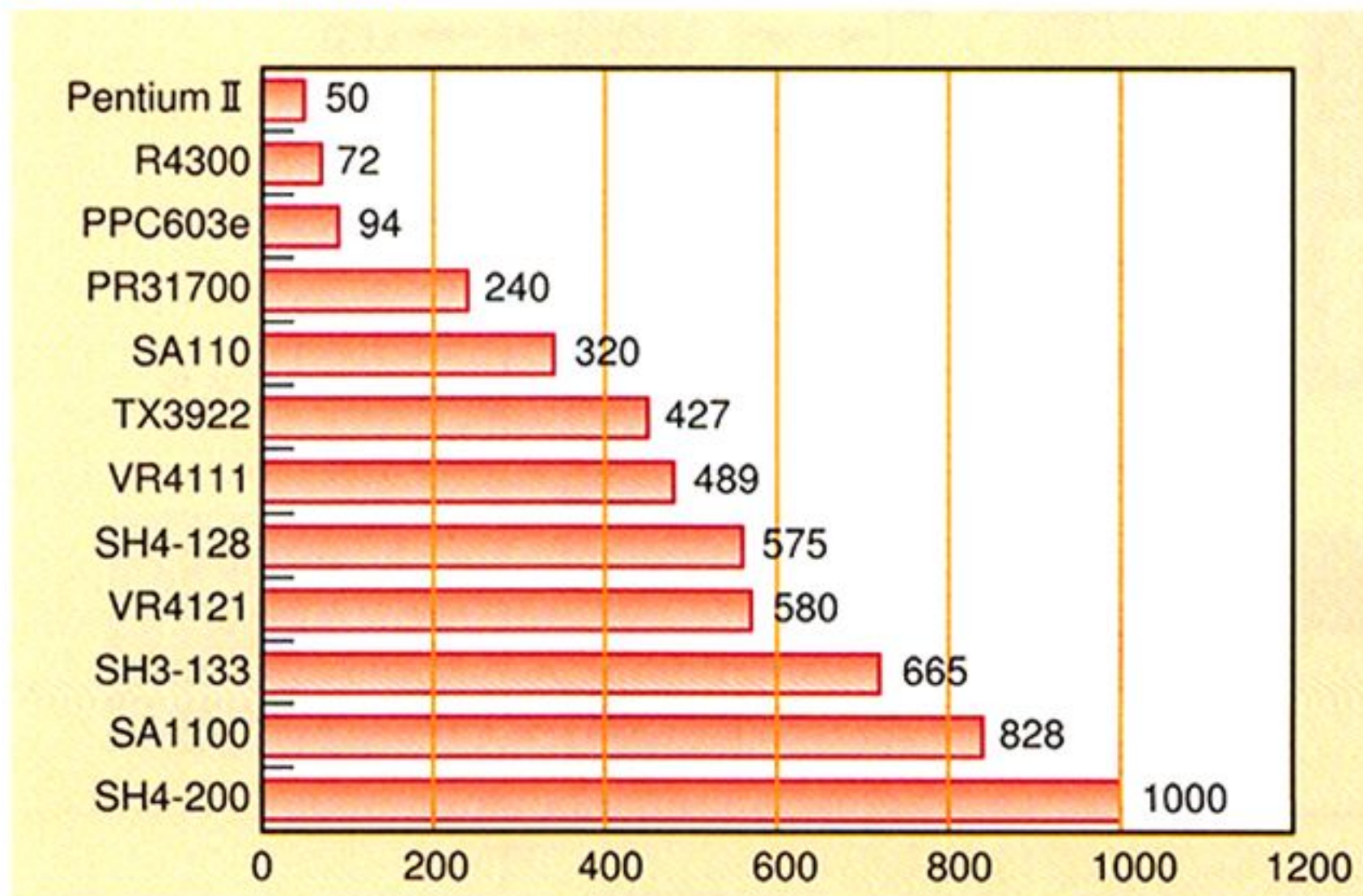
WindowsCEに採用されているCPUのMIPS/WがPentiumIIなどと比べてはるかに高い値になっていることに注目してほしい。もっとも，カラー画面の製品についてはLCDで消費される電力が非常に大きいので，CPUが多少電気食いでも電池寿命にさほど影響はない(短いという意味では変わらない)。逆にモノクロ画面ではCPUの消費電力が直接電池寿命に影響する。CEマシンはいまでこそカラー画面が全盛だが，PalmOS対抗としてこの先モノクロ版のPsPCが登場することは大いに予想できる(現存のモノクロ版PsPCの電池寿命はかなり短い)。そのときになれば再度消費電力がクローズアップされてくるに違いない。

WindowsCEは本来はモノクロ画面で登場し，カラー化に対応することでPalmOSと差別化を狙った感がある。そのPalmもカラー化されているいま，状況は混沌としている。モノクロでいいから長時間使用したいユーザーと，なにがなんでもカラーでなくちゃというユーザーに二分されるのかもしれない。ただし，全体の流れとしてはカラー化に進んでいると思う。そもそもカラーかモノクロかというのは，なにを目的とするかによる。名刺管理や住所録のようなPIM(Personal Information Management)はモノクロで十分だし，インターネット接続やGPS(Global Positioning System)と連動した地図情報の利用などを目的とすればカラー化は必要であろう。

消費電力に関して興味深い報告をひとつ。一般にCPUの動作周波数を上げると消費電力も増大する。しかし，CEマシンにおいてはそれは必ずしも正しくない。動作周波数が速いと周辺回路が動作する時間が短くなるため，結果として消費電力が少なくなる場合もある。ただし，これは通常動作時の消費電力の話である。電池寿命に効いてくるのは主としてサスペンド(待機)時の消費電力である点は変わらない。

WindowsCE機の電池寿命はどの程度が望ましいのであろうか。基本的に携帯するものであるが，さすがに自宅での使用時はACアダプタを利用するであろう。PsPCなどは携帯電話のようにクレイドルに置いて充電しているはずである。これを考えると最初の上限は，半日，つまり10時間程度にあると思われる。その次が小旅行時などでの使用で1週間，その次が1カ月というところであろうか。まさに携帯電話の電池寿命と同じである。まあ，CEマシンとして当面の目標は10時間であろう。カラー画面で，しかも

図1　各CPUのMIPS/W

CPU	MIPS/W
Pentium II	50
R4300	72
PPC603e	94
PR31700	240
SA110	320
TX3922	427
VR4111	489
SH4-128	575
VR4121	580
SH3-133	665
SA1100	828
SH4-200	1000

高負荷状態で使用して10時間電池が持つというCEマシンはまだ発売されていない(と思う)。表4に最新(1999年末時点)のWindowsCEマシンの電池寿命(カタログ値)を示す。カラー機は10時間(半日)，モノクロ機は25時間(1日)が電池寿命の目標であることがわかる。装置の構成にもよるので電池寿命が即CPUの消費電力を表しているわけではない。しかし，CPUの消費電力が大きいと電池寿命が短くなるのは自明である。ちなみにDragonballの消費電力は7mW程度。この電力でモノクロ表示なら電池寿命が2週間というのも納得。ただし，2000年2月に発表されたPalmのカラー版であるPalm IIIcでは電池寿命が9時間弱になった(Palmのサイトではクレイドルと兼用で2週間以上の電池寿命と意味不明な記述があるが)。やはりカラーだと10時間程度が限界なのか。

CrusoeのLongRun，AMDのPower Now！やPentiumIIIのSpeedStepなどPCの世界でも低消費電力化技術が花盛りである。しかし，その場合の電力も1W程度でWindowsCE用のCPUとは比較にならないほど大きい。AMDの発表ではSpeedStepは7W程度，Power Now!は3W程度だという。それに加え，SpeedStepによって電池寿命が10〜20%(Power Now!では30%)長くなるが性能はがた落ちということで，ユーザーはSpeedStepモードを無効化して使用する可能性が高いという予測もある。WindowsCEをはじめとするPDA用のCPUとPCの用のx86CPUとの間にはまだ高い壁があるのは確かである。2000年4月，NS(National Semiconductor)がx86互換のGX1というチップを発表した。性能は不明だが，消費電力が1W以下で，WindowsCE用のCPUと競合していると聞いている。最近(2000年6月)ではCrusoe搭載のノートPCの発表が盛んに行われているが，性能はともかく，その約1Wという消費電力は魅力的なのかもしれない。

● 周辺機能の内蔵

製品をコンパクトに仕上げるためには部品点数は少ないほうがよい。CEマシンにおいても周辺機能を1チップに内蔵するCPUが好まれる。といっても，多くのCPUに内蔵されている周辺機能は，メモリコントローラ，タイマ，DMAといった基本的なものである。装置に特化した機能は各メーカーがASICを起こして対応している。CPU側で提供する基本機能としては，最近ではPCIバスやUSBインタフェイスの内蔵が流行であろうか(どちらも電気食いなのだが)。また，タッチパネルやオーディオ用のためのAD/DAコンバータといったアナログ系機能は通常別チップで供給される。全体として，2〜3チップ構成でシステムを構成することが多い。表5に主要なCPUの内蔵周辺機能の一覧を示す。

ところで，WindowsCE対応のCPUのプレスリリースでは「WindowsCEに最適な周辺機能を内蔵」という言葉をよく見かけるが，具体的にどういうものかはよくわからない。表5に示す周辺機能がどれも似通っているということは，多分これらの機能を表しているのだろう。

表4　最新WindowsCEマシンの電池寿命

メーカー	製品名	CPU	電池寿命
NEC	Mobile Gear II MC/R530	VR4121-168MHz	4.5〜10時間
NEC	Mobile Gear II MC/R430	VR4121-168MHz	4.5〜10時間
NEC	Mobile Gear II MC/R330*	VR4121-131MHz	25時間
JVC	Inter Link MP-C101	TX3922-129MHz	6時間
富士通	INTERTOP CX310	VR4121-168MHz	8.5時間
SHARP	Telios HC-AJ2	TX3922-129MHz	8時間
SHARP	Telios HC-VJ1C	TX3922-148MHz	10時間
日立	PERSONA HPW-50PA	SH3-100MHz	8時間
日立	PERSONA HPW-600JCM	SH4-128MHz	9時間
HP	Jornada 680 (PostPet)	SH3-133MHz	9時間
HP	Jornada 690	SH3-133MHz	9時間
HP	Jornada 820	SA1100-190MHz	13.5時間
Compaq	AERO 8000	SH4-128MHz	7時間
Compaq	PRESARIO 213	VR4111-70MHz	10時間
CASIO	CASSIOPEIA E-503	VR4121-131MHz	6時間
CASIO	CASSIOPEIA E-65*	VR4111-69MHz	25時間
高木産業	PT-C01	VR4111-70MHz	8〜10時間
高木産業	PT-M01*	VR4111-70MHz	20時間

(注)*印はモノクロマシン。

表5　各CPUの周辺機能

SA1100	SH3	SH4	TX3922	VR4121	VR4122
MEMC	MEMC	MEMC	MEMC	MEMC	MEMC
シリアル	シリアル	シリアル	シリアル	シリアル	シリアル
IrDA	IrDA	IrDA	IrDA	IrDA	IrDA
RTC	RTC	RTC	RTC	RTC	RTC
Timer	Timer	Timer	Timer	Timer	Timer
DMAC	DMAC	DMAC	DMAC	DMAC	DMAC
INTC	INTC	INTC	INTC		
	A/D,D/A			A/D,D/A	
GPIO			GPIO	GPIO	GPIO
PCICMA			PCICMA		
					PCI

● 各CPUの実際

各社のCPUの構成を見てみよう。図2にVR4121のブロック図(http://www.ic.nec.co.jp/micro/product/vr/vr4100series/index.htmlより), 図3にTX3922のブロック図(http://www.semicon.toshiba.co.jp/noseek/jp/pr/txfm.htmより), 図4, 図5にSH3, SH4のブロック図(ユーザーズマニュアルより), 図6にSA1100のブロック図(http://www.pc.com/design/strong/sa1100.htmより)を示す。

上記のサイトには, TX3922とSH4に関して, WindowsCEのシステム構成図が掲載されているので, それぞれ, 図7, 図8に示しておく。

ベンチマークテスト

世の中ではいろんな種類のWindowsCEマシン(H/PC, PsPC)が販売されている。購入を考えている人にとってはなにか指針がほしい。そこでいくつかの団体が標準ベンチマークを定めてCEマシンの性能比較を行っている。ここではCPUの種類に注目して, 各CPUのベンチマークテストの結果を見てみよう。

WindowsCE用のベンチマークテストとしてはDBench, bBenchの2つが特に有名である。これらのベンチマークテストは, どれもCPUの演算性能とグラフィックの描画性能に注目している。描画性能はグラフィックコントローラに依存する部分が多いので, ベンチマーク結果が, 即CPUの性能ではないことに注意が必要である。しかも, これらのベンチマーク結果は, 必ずしも体感性能を表していない。

コンピュータ専門誌で, ときどき, WindowsCEマシンのベンチマークテストが行われているが, それらはどれも体感性能を重視している。このため, DBenchやbBenchの結果と食い違いが生じている。実際に使用する立場では体感性能のほうが重要なのは明らかである。しかし, 搭載されているCPUブランドでCEマシンを選択をした人にとっては, 高性能なCPUのくせに体感性能が悪い場合は, 複雑な気分であろう。そもそも, ベンチマークテストと体感性能に差が出るのは, WindowsCE上でのアプリケーションプログラムがなぜかキャッシュにヒットしない(と思われる)のが最大の原因である。これはマイクロソフトの陰謀か。だから, 動作周波数よりもバス速度の速いCPUを使用するほうが有利になる。

● DBench

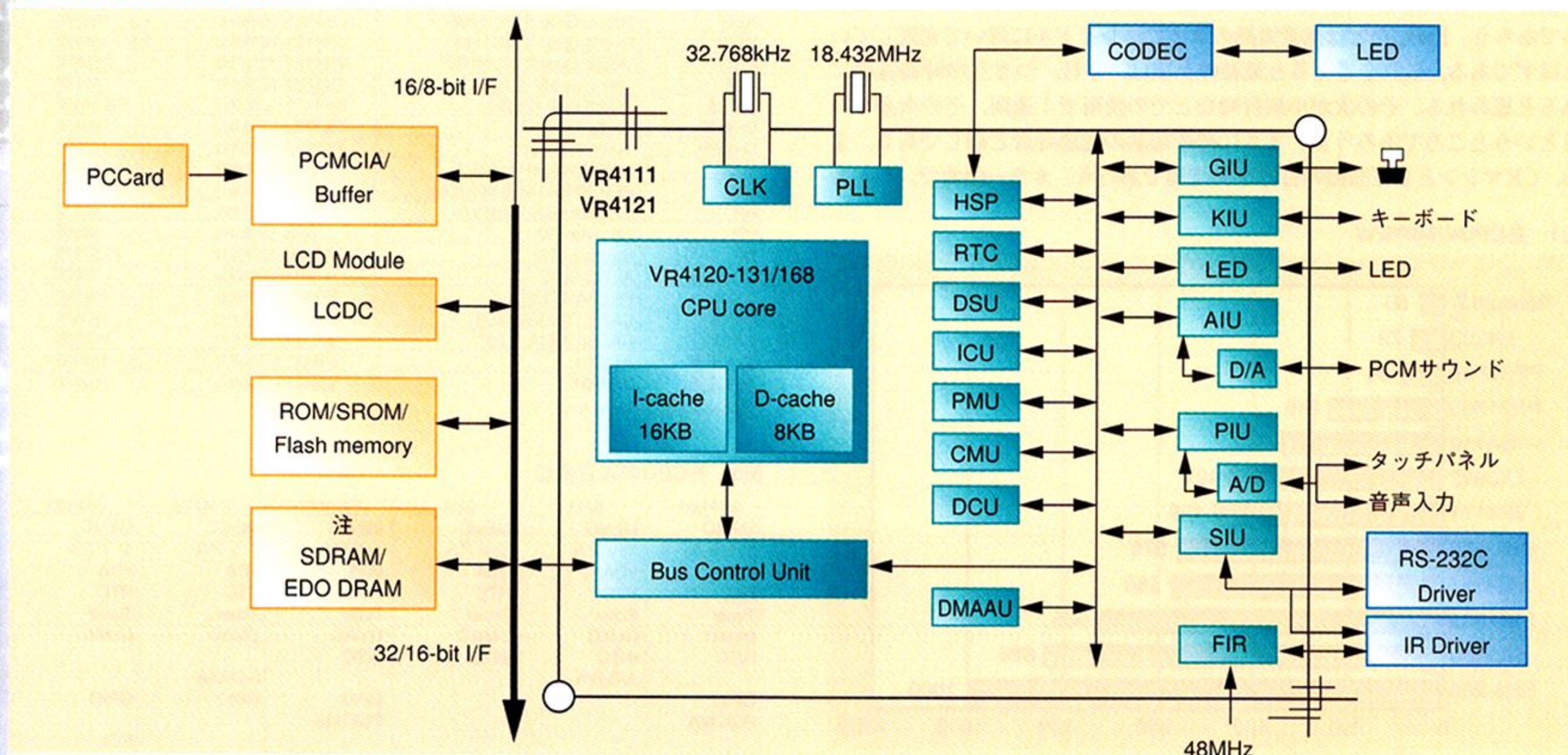
DBenchとはWindowsCE情報では定評のあるWebサイトであるWindowsCE FAN(http://wince.ne.jp/)で利用しているベンチマークテストである。これは, D Collectionsを主催する傍島康雄氏作成のベンチマークソフトである。以下に示す6種類の項目からなる。具体的な内容は, 作者に連絡を取ればソースを公開してもらえるそうだが, それには興味がないので, 実行の様子からの想像である。間違っている可能性は十分ある(たいした問題ではない)。H/PC ProのCPUに着目した実行結果を図9に示す(http://wince.ne.jp/review/frame.asp?/review/katsuo/bench/index.htmから値を引用)。CPUとWindowsCEマシンの対応は次のようになっている。ただし, R530とJornada680に関しては画面サイズがHalf VGAなので単純には比較できないので注意を要する。また, DBenchはタイマの処理がまずく正確な性能を測定できないことが指摘されている。たとえば, TX3922の129MHz品と148MHzの性能がほとんど同じに見えてしまう。

SH4-128MHz	日立 PERSONA HPW-600JC
VR4121-131MHz	富士通 INTERTOP CX300
TX3922-129MHz	シャープ Telios HC-AJ1

図3 TX3922のブロック図

図2 VR4121のブロック図

VR4121-168MHz	NEC MobileGear II MC/R530
SH3-133MHz	HP Jornada 680
TX39220-148	シャープ Telios HC-VJ1C

CPU単体の性能がわかるのは「Int calc」と「Double calc」くらいであろうか。結果を見るとSH4の性能が頭ひとつ抜けている（Half VGAマシンを除く）ように思えるが，よく見ると性能を稼いでいるのはグラフィックの性能である（CPU性能とは無関係）。「Int calc」と「Double calc」の結果のみに注目するとSH4の性能は結構悲惨である。この観点では，VR4121-168MHzがもっとも性能がよく，続いて，TX3922とVR4121-131MHzが横並びである。経験的にDBenchはキャッシュにヒットしやすいベンチマークテストである。その点でCPUのバス速度の違いは結果に表れてこない。その意味で，体感性能との相関は少ないが，CPU本来の（パイプライン効率など）の性能比較には役立つ（でも，実用性は疑問）。

(1) Int calc

これは整数演算の性能をテストする。主として足し算の性能をテストしているらしいが，逆アセンブルしたところ，乗算命令をループしているようである。このため，乗算と分岐が高速なCPUに有利である。システ

図4　SH3のブロック図

〈記号説明〉

ADC	A/D変換器
BSC	バスステートコントローラ1
BSCP	バスステートコントローラ2
CACHE	キャッシュメモリ
DCN	キャッシュメモリコントローラ
CPG/WDT	クロック発振器/ウォッチドッグタイマ
CPU	中央処理装置
DAC	D/A変換器
DMAC	ダイレクトメモリアクセスコントローラ
INTC	割り込みコントローラ
MLT	乗算器
MMU	メモリマネージメントユニット
RTC	リアルタイムクロック
SCI	シリアルコミュニケーションインタフェイス（スマートカードインタフェイス付き）
IrDA	シリアルコミュニケーションインタフェイス（IrDA付き）
SCIF	シリアルコミュニケーションインタフェイス（FIFO付き）
TLB	アドレス変換バッファ
TMU	タイマユニット
UBC	ユーザーブレークコントローラ

図5　SH4のブロック図

〈記号説明〉

CCN	キャッシュ&TLBコントローラ
BSC	バスステートコントローラ
CPG	クロック発振器
OMAC	ダイレクトメモリアクセスコントローラ
FPU	浮動小数点演算ユニット
ITLB	命令TLB（Translation Lookaside Buffer）
UTLB	共用TLB（Translation Lookaside Buffer）
RTC	リアルタイムクロック
SCI	シリアルコミュニケーションインタフェイス
SCIF	FIFO内蔵シリアルコミュニケーションインタフェイス

図6　SA1100のブロック図

図7　TX3922のシステム構成例

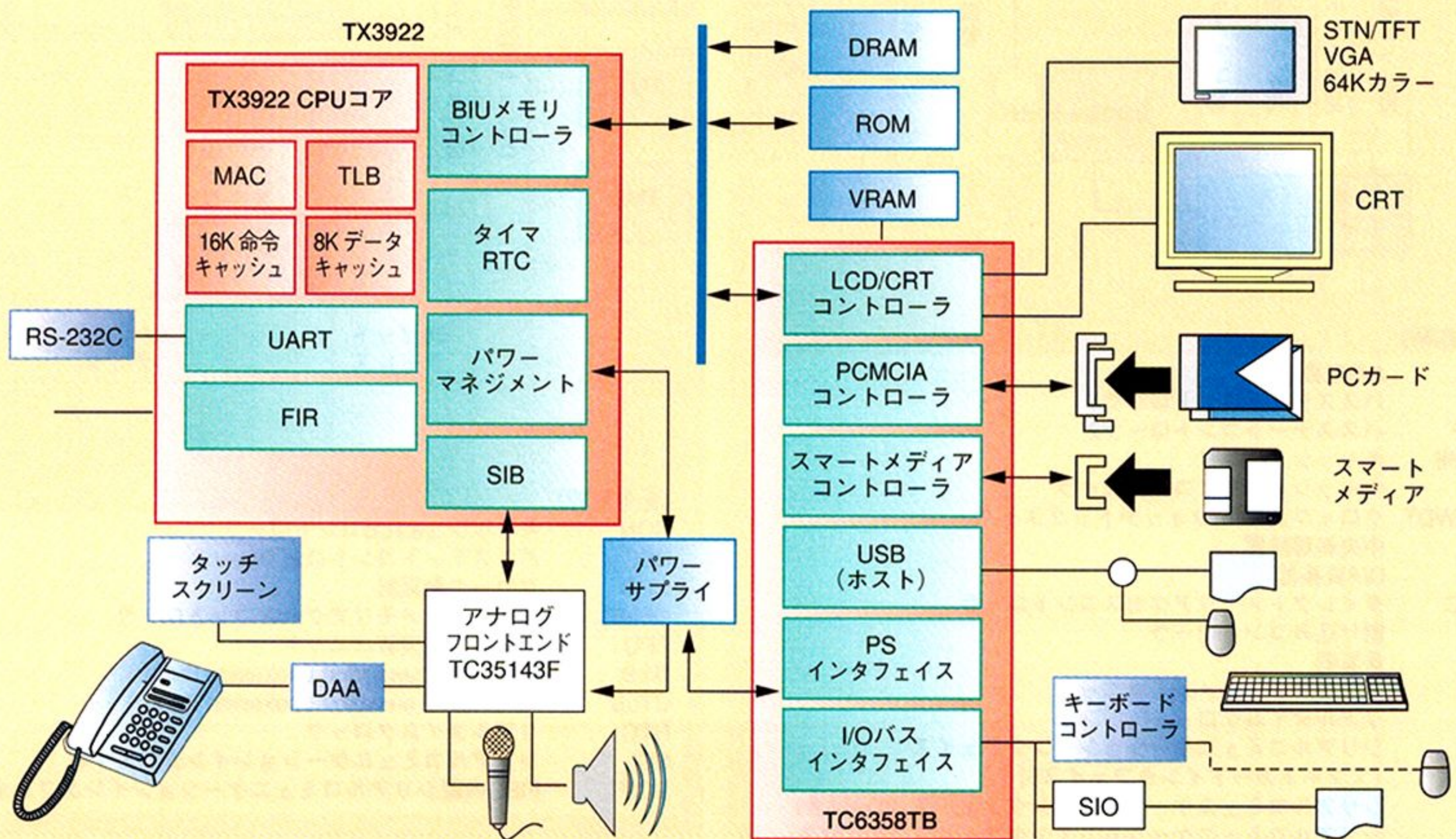
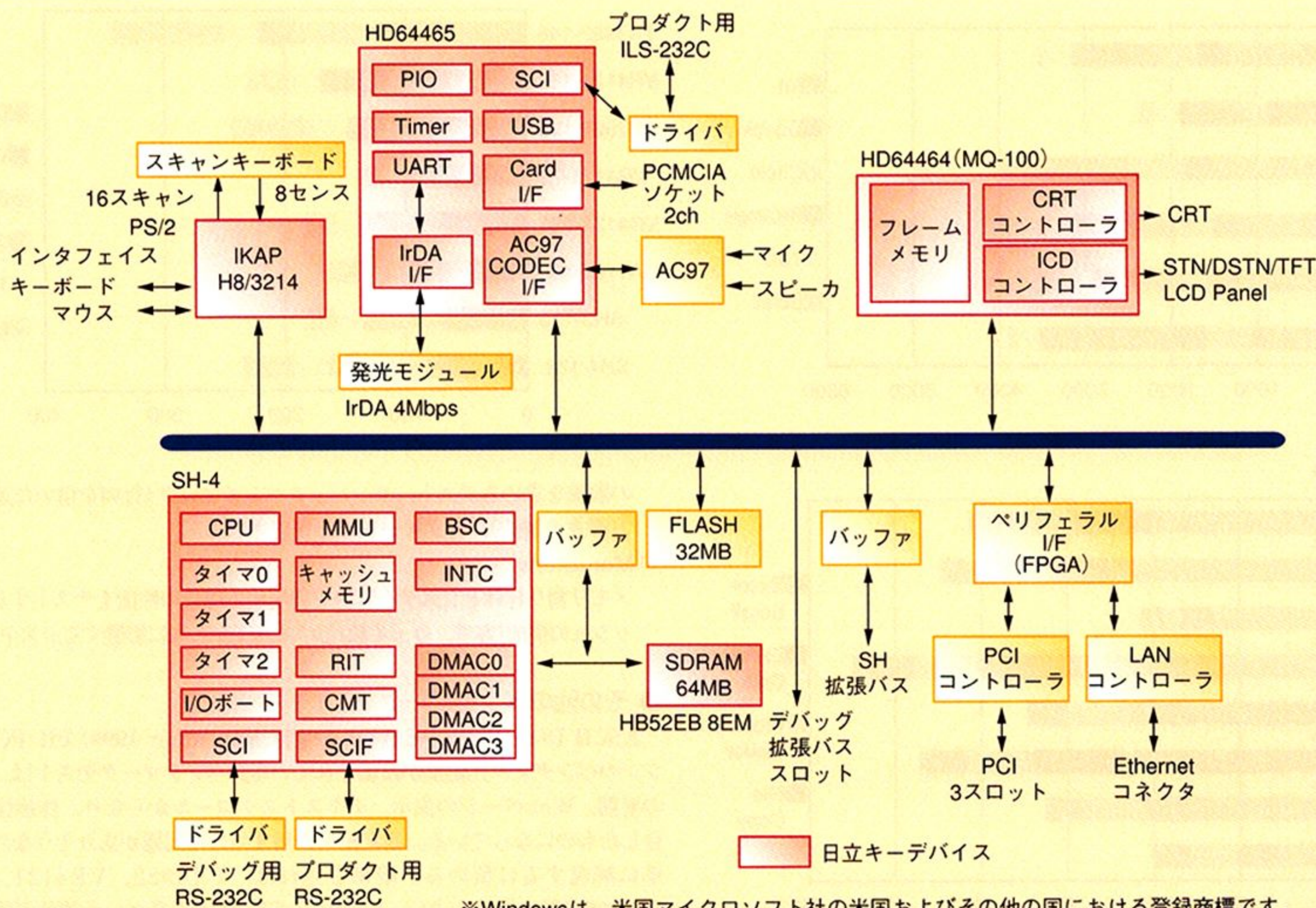



図8　SH4のシステム構成例

※Windowsは，米国マイクロソフト社の米国およびその他の国における登録商標です。
※Ethernetは，米国ゼロックス社の登録商標です。
※MQ-100は，米国MediaQ社との契約により，日立で製品化したLSIです。

ム性能として乗算性能が支配的という着眼点がいい。

(2) Double calc

これは浮動小数点演算をソフトウェアでエミュレートした場合の演算性能をテストする。名称から，なんらかの倍精度演算をエミュレートしていると思われる。エミュレートルーチンは結構巨大なので，キャッシュのヒット率(つまりは容量)が多少影響する。

(3) Circle Draw

画面上に多くの円形を色付きで描画する。円形描画性能をテストする。

(4) Rectangle Draw

画面上に多くの長方形を色付きで描画する。矩形領域の塗りつぶし性能をテストする。グラフィックアクセラレータの性能が非常に影響する。

(5) Text Draw

画面のウィンドウ内に文字を描画する。テキスト描画性能のテスト。

(6) Scroll

小さなウィンドウを表示し，それを左右上下にスクロールさせる。スクロール速度のテスト。

● bBench

BSQUARE社がWindowsCEシステム開発のために提供しているbUSEFULというソフトウェアパッケージの中にbBenchというベンチマークがある(http://www.bsquare.com/products/p_buseful.htmを参照のこと)。これは以下に示す6本のベンチマークテストから構成され，BSQUARE社独自の重み付けによって結果が表される。少し前までは，このbBenchの結果がシステム性能をよく表すといわれていた。現在は体感性能とのギャップが指摘されている(それでもDBenchよりは遥かに相関性が高い)。しかし，CPU自体の性能比較をするためには意味がある(実際にはセットとしての性能が重要なのでその行為自体は無意味なのだが，本稿はCPUにスポットを当てているので)。

このベンチマークテストのH/PC Proの結果を図10に示す(http://pda.tucows.com/wincelair/wincespeed.htmから値を引用したが現在はリンク切れ)。なお，CPUとWindowsCEマシンの対応は次のようになっている。米国のサイトなので，ベンチマーク対象は米国で発売されているマシンに限定されているが使われているCPUは日本のCEマシンと大差ない。VR4121の168MHzおよびTX3922の148MHz(2)の値は，それぞれ，知人にMobile Gear II MC/R730とTelios HC-VJ1Cの結果を測定してもらった結果である。

SH4-128MHz	Compaq Aero 8000
SH3-133MHz	HP Jornada 680
SA1100-129MHz	HP Jornada 820
VR4121-131MHz	NEC MobilePro 770
VR4121-168MHz	NEC Mobile Gear II MC/R730
VR4111-70MHz	Vadem Clio
TX3922-129MHz	Sharp Mobilon PV-5000
TX3922-148	Sharp HC-VJ1C

bBenchではGraphStone以外はCPU性能に直接関係すると思われる(MemStoneはメモリアロケーションを行うのでOSの性能も多少影響するが)。結果を見ると，TX3922の性能がもっともよく(HC-VJ1Cの値は異常によすぎる)，続いてSH4，VR4121，SH3の順である。これによるとSH4の性能はかなりよい。DBenchで良好な結果を示していたVR4121の168MHz品もbBenchではそれほどいい値を示していない。これはTX3922に比べてWhetstoneとMemstoneの値が低すぎるのが原因である。おそらく，キャッシュの構成(ダイレクトマップが2ウェイセットアソシアティブか)が影響しているのであろう。

Windows(CEに限らず)でのプログラミングスタイルは，システムコールで4Kバイトなり8Kバイト単位のメモリ領域を確保して，そこを作業域に使うことが多いのでキャッシュを参照するアドレスのインデックスが重な

図9　DBench

図10　bBench

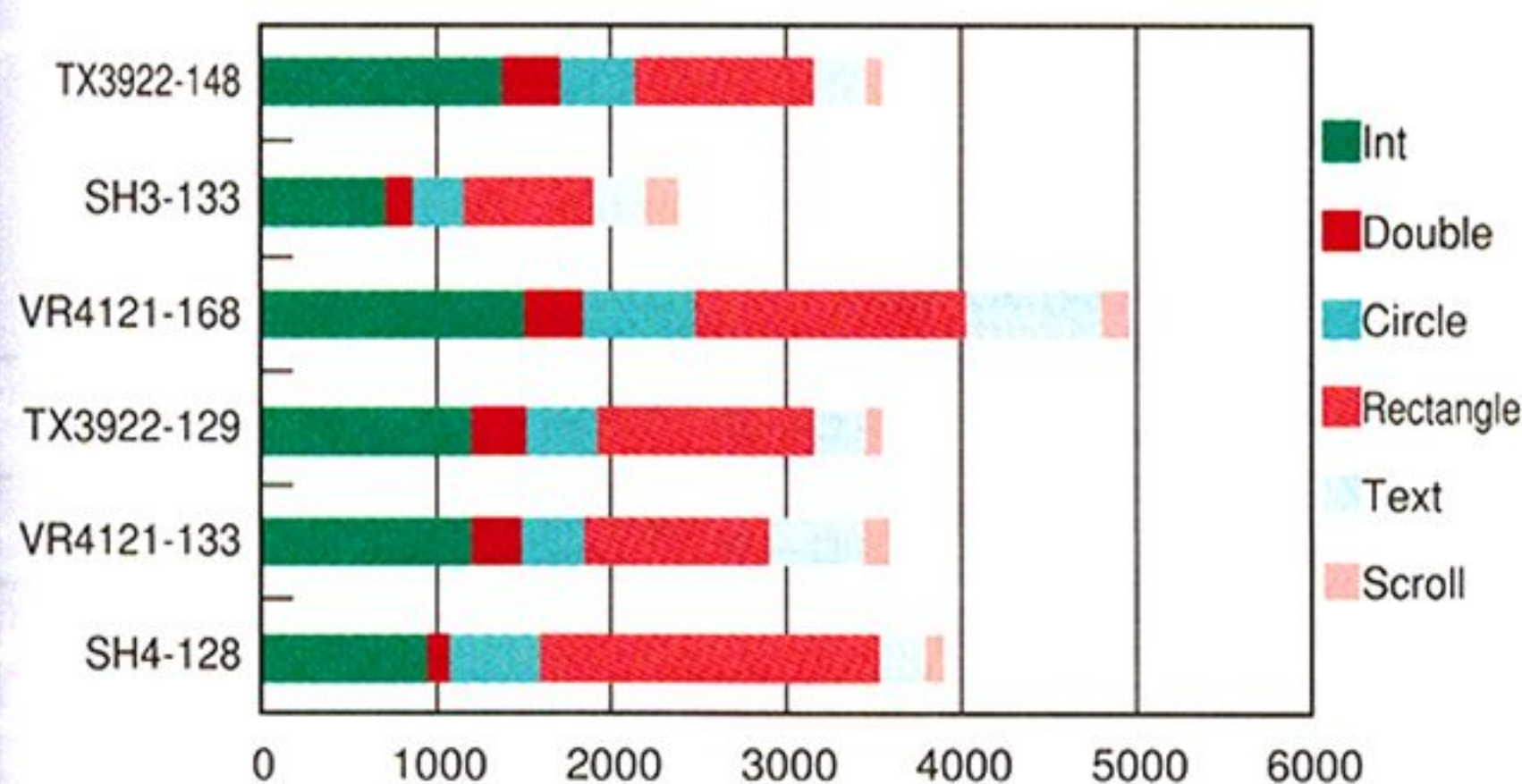
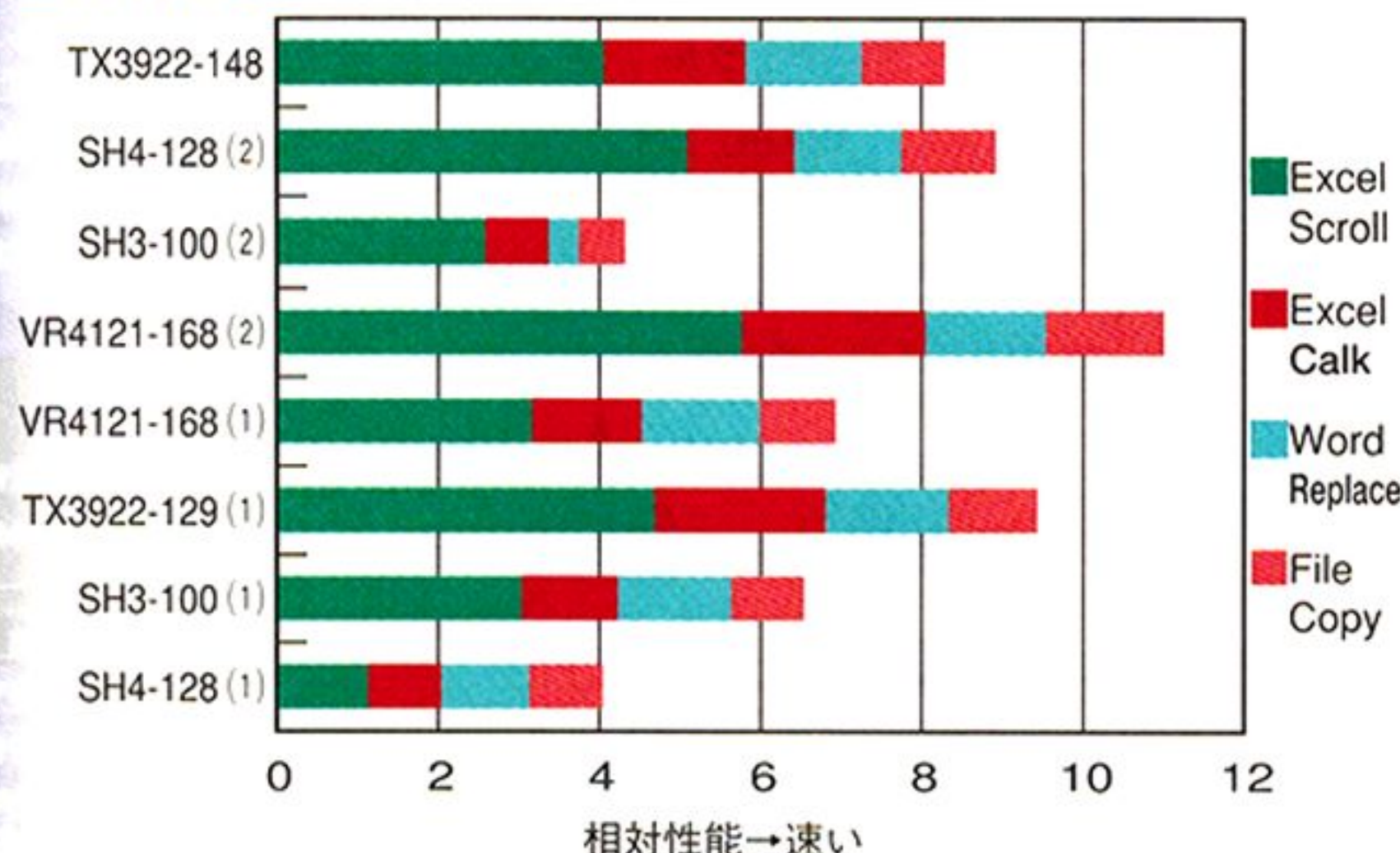



図11　体感性能

ることが多い。このような場合はキャッシュのウェイが多いほど有利である。なお，ダイレクトマップキャッシュであるSH4の値が思いのほかいいのはスーパースカラの恩恵だろう。DBenchで株を下げたSH4の面目躍如である。同じTX3922を使用していても，HC-VJ1Cの値が，PV-5000に比べて極端にいいのはバス速度の違いと推測される（動作周波数も向上しているが）。いずれにしても動作周波数が性能を決める最大要因でないことがわかる。周波数競争に躍起になっているIntelやAMDに教えてあげたい。

(1) Dhrystone

有名な整数性能を測定するベンチマークテストである。bBenchでは速度よりもプログラムサイズに注目した最適化をしてある。たいていのCPUならキャッシュに入り切るはずであるが，なぜか性能がCPUのバス速度にも影響する。データ領域をシステムコールで確保するためだろうか（上述のインデックスの重なりが生じる？）。

(2) Whetstone

有名な浮動小数点性能を測定するベンチマークテストである。bBenchでは浮動小数点演算をソフトウェアでエミュレートした場合の演算性能をテストする。命令キャッシュの容量にかなり影響されるようである。これは，浮動小数点演算のエミュレーションプログラムが巨大なことを意味する。

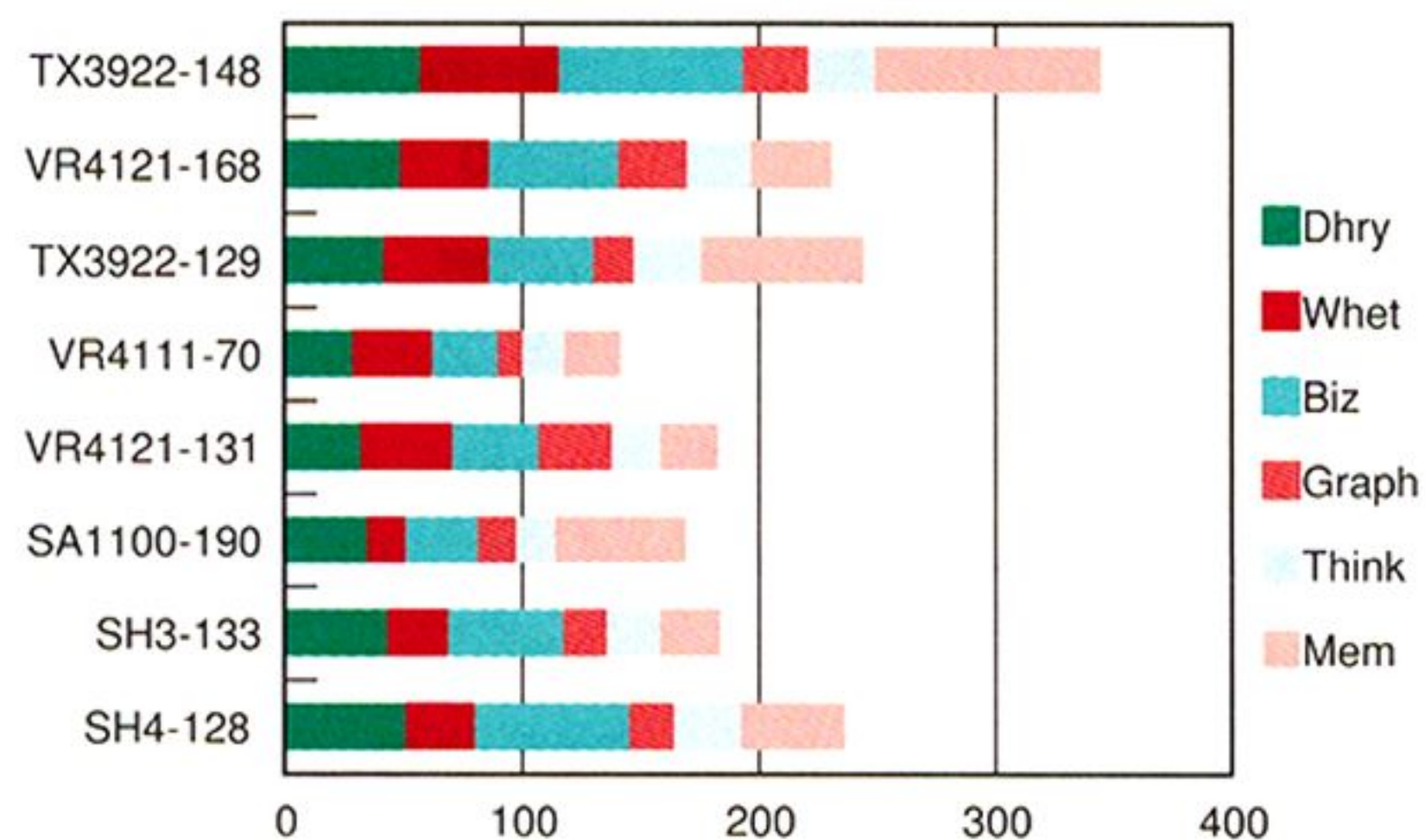
(3) Bizstone

データベースとワードプロセッシング操作の性能をテストする。具体的には，QuickSort，Distributed Sort，Heap Sortという3種類のソートプログラムで構成される。キャッシュのヒット率があまりよくないのでバスのスピードが多少影響する。

(4) Graphstone

15カテゴリからなるGDI（Graphics Device Interface）操作の性能をテストする。簡単にいえばグラフィック性能のテストである。

(5) Thinkstone

人工知能関係のプログラム性能をテストする。πの10番目から5000番目の数値を求めるテスト，リンパックベンチマーク（行列を用いた連立1次方程式の解法），nクイーンから構成される。

(6) MemStone

メモリ割り付けとシステム資源を利用する場合の性能をテストする。キャッシュの構成（容量，ウェイ数）とバス速度がもろに影響するテストである。

● その他のベンチマーク

ASCII DOS/V ISSUE（VOL.5, #11 November 1999）でH/PC Proマシンのベンチマーク結果が掲載されている。ベンチマークテストは，データの展開，Webページの表示，テキストスクロールからなり，体感性能に注目したものになっている。測定値を引用するのは問題がありそうなので，簡単に解説するに留める。結果を見れば，TX3922，VR4121，SH4，SA1100の順に性能がよい。WindowsCEはMIPS系という神話が築かれつつあるか。それにしても，bBenchでは良好な結果を残していたSH4はやはり悲惨である。SH4のファンには意外な結果かもしれない。筆者はひとえに命令キャッシュの容量が8Kバイトと少ないことに理由があると考える。また，SA1100は190MHzという高い動作周波数でありながらまったくいいところがない。HP社のJornadaシリーズはSH3を採用して好評を博してきただけに，もしかしたら，SA1100の採用は汚点だったかもしれない。そのせいかどうかは知らないが，最近のジェームズボンドの映画に登場した（はずの）Jornada430seでは再びSH3が採用されている。この映画は未見なので詳細は知らないが，なんでも爆弾の起爆装置の解除に失敗するツールとして登場するとか（おお，イメージダウン）。

CNETが行ったキーボード付きのH/PCのベンチマーク（`http://www.computers.com/reviews/comparative/guide/0,28,0-20-733094-733148,00.html?tag=st.co.pdtl.dir.perf`）というのもある。これは，WindowsCEに付属しているPocketExcel，PocketWord，Contactsを用いてデータアクセスの速度を比較している。これも体感性能を重視したベンチマークといえる。結果は，

1. SH3-133MHz（HP Jornada 680）
2. SA1100-190MHz（HP Jornada 820）
3. VR4121-131MHz（NEC MobilePro 800）
4. TX3922-129MHz（Sharp Mobilon Pro PV-5000）
5. VR4111-90MHz（Vadem Clio C1000）
6. VR4121-131MHz（IBM WorkPad z50）
7. SH3-80MHz（Novatel Wireless Contact）
8. TX3912-75MHz（Sharp Mobilon TriPad PV-6000）

という順番になっている。SH3が1番という，ほかのベンチマークとは少し異なる結果が出ている（このサイトではSH3をVR4121と間違えているが，190MHzのCPUより133MHzのCPUのほうが高速な点に驚いている）。実際，Jornada 680には固定ファンも多く，体感性能の高さは多くの人が認めている。メインメモリの容量が32MバイトになったJornada 690もかなりの評判だ。

わがソフトバンクのDOS/Vmagagine誌上でも2000年1月15日号で，最新（1999年末時点）WindowsCEマシンの性能比較を行っている（pp.216-

217)。これらのテストはExcelスクロール速度テスト，Excel再計算速度テスト，Word置換速度テスト，ファイルコピー速度テスト，重負荷バッテリ起動時間からなっている。CPUの性能にはあまり関係ないバッテリ起動時間以外のテストについて，それぞれ，Aero8000の性能を1としてプロットしたものが図11である。図11ではCPU名と動作周波数で示してあるが，搭載機種との対応は次のようになっている。

SH4-128(1)	Compaq Aero8000
SH3-133(1)	Jornada680
TX3922-129(1)	JVC InterLink MP-C101
VR4121-168(1)	Fujitsu INTERTOP CX310
VR4121-168(2)	NEC MobileGear II MC/R530
SH3-100(2)	Hitachi PERSONA HPW-50PA
SH4-128(2)	Hitachi PERSONA HPW-600JCM
TX3922-129(2)	Sharp Telios HC-AJ2

図11によると，VR4121とTX3922がよい値を出している。SH3はちょっと性能不足というところ。SH4に関しては，Aero8000とPERSONAでの性能差がありすぎる(Compaqめ，手を抜いたな)。まあ，CPU単体の性能だけではマシンの体感性能を推測できないといういい例かもしれない。システム構成に依存する部分が大きいということである。

● 結局，最強のCPUは

上述のベンチマークテストの結果を見ると，最強のCPUはTX3922ということになる。次いでVR4121であろうか。SH4に関しては速いのか遅いのかよくわからない。ベンチマークテストに登場していないSA1110，VR4122の性能が気になるところである。

各社のCPU戦略

WindowsCE用のCPUは今後どのようになっていくのか。メーカー各社の戦略を俯瞰してみよう。注目は，WindowsCE1.0の時代から開発を続けてきているNECと日立である。ただし，以下の文章は想像が占める部分が多いことをお断りしておく。

● 東芝

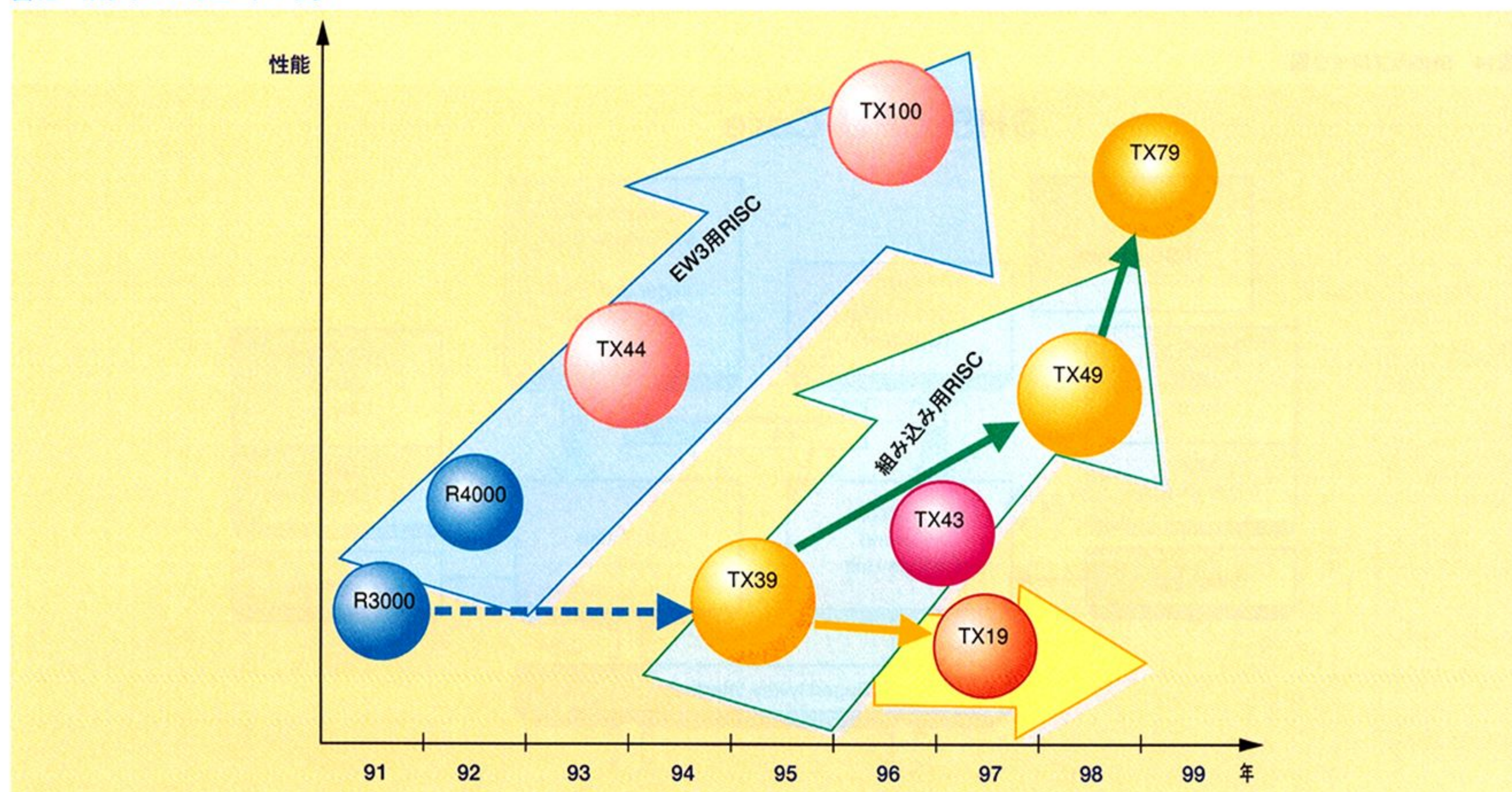
TX3922という，最強のCPUを製造している東芝であるが，いかんせん，あとが続かない。TX3922の148MHz版は，ずいぶん前にアナウンスされたにもかかわらず，依然開発中となっているし，後継のTX49シリーズについても発売したという噂は聞かない(一部にはR3000系とR4000系で互換性がないので採用されないという噂もあるが真実は不明)。そもそもTX3922は166MHz動作として発表されたはずである。もしかしたら，今後WindowsCE用のCPU供給をやめてしまうのではという危惧もある。まあ，東芝にはPS2用のEmotionEngineがあるのでそれでいいのかもしれない。EmotionEngine自身の周辺機能はSCEの特許そのものなので東芝の自由にはならないと思われる。しかし，CPUコアは東芝オリジナルのはずなので，それに新しい周辺機能を付加した商品展開は当然考えられる。TX39シリーズが150MHz動作を達成しようと四苦八苦しているのと対照的に，EmotionEngineはすでに300MHzで動作しているのが強みである。ただし，CPUコアが250MHz動作時に5W強という消費電力はちょっと大きすぎるかもしれないが，時が解決するだろう。

図12に，TXシリーズのロードマップを示しておく(`http://www.semicon.toshiba.co.jp/noseek/jp/pr/txfm.htm`より)。これによると，そろそろTX79が発表になってもよい頃なのだが。さて，このTX79であるが，日本の東芝のサイトにはなんの説明もない。しかし，米国のサイト(`http://www.toshiba.com/taec/`)では128ビットRISCと説明されている。TX79がロードマップに載ったのは1999年の初頭で，EmotionEngineがISSCC'99で発表される直前である。このとき一部の掲示板でTX79がPS2のCPUとしてクローズアップされた。もしかしたらTX79は本当にEmotionEngineのCPUコアなのかもしれない。だったら面白いのになあ。

なお，2000年になって東芝のWindowsCE関係のページ(`http://www.semicon.toshiba.co.jp/noseek/jp/pr/wincefm.htm`)が更新された。そこではTX4955のシステムが紹介されている。東芝はまだ死んでいないのか，と思っていたところ，2000年2月のCeBITでシャープのTeliosの後継機であるHC-7000が発表された。CPUは148MHzのTX3922らしい。でも，発売日未定(8月という噂も)というのがちょっと気になる。このHC-7000の日本向け版がHC-VJ1C(3月10日発売)である。

● NEC

NECは堅実にWindowsCE用のCPUを供給し続けているように思える。VR4121の168MHz版を搭載したMobileGearIIやINTERTOP CX310が最

図12 TXシリーズのロードマップ

近(1999年末)発表されたし，後ろにはキャッシュ容量を倍増したVR4122が控えている。VR4122といえばMIPS/Wの値でSA1110を上回ったということで注目されているCPUである。さすがNECといったところか。とはいえ，SH5や新StrongARMのWindowsCEマシンに対抗するためには，VR4122では(性能的に)やや力不足の感は否めない。これらのCPUに匹敵する性能を持つ次期製品が期待される。ところで，ISSCC2000でMP98という，4個のスーパースカラプロセッサを1チップに集積した，1000MIPSの性能のCPUが紹介されたが，いかんせん，アーキテクチャがV800である。MIPS系ならWindowsCEへの展開が期待できたのに，MMUのないV800では期待薄である。

なお，WindowsCEのPsPCに関しては，最大のシェアを誇るカシオのCASSIOPEIAのEシリーズを筆頭に，CompaqのPresario213，IBMのWorkPad z50，VademのClio，EverexのFreeStyleと，CPUはNECの独壇場である。カシオ以外はPsPCから撤退しているようであるが，逆にカシオの独占状態が続くものと思われる(E-500の海外版であるE-100は大人気であるという)。実際，WindowsCE3.0でもすでに3社(Compaq，CASIO,Symbol)がNECのCPUの採用を表明している。

また，H/PCに関しても，東芝の元気がないので，MIPS系のCPUに関し

図13 VRシリーズのロードマップ

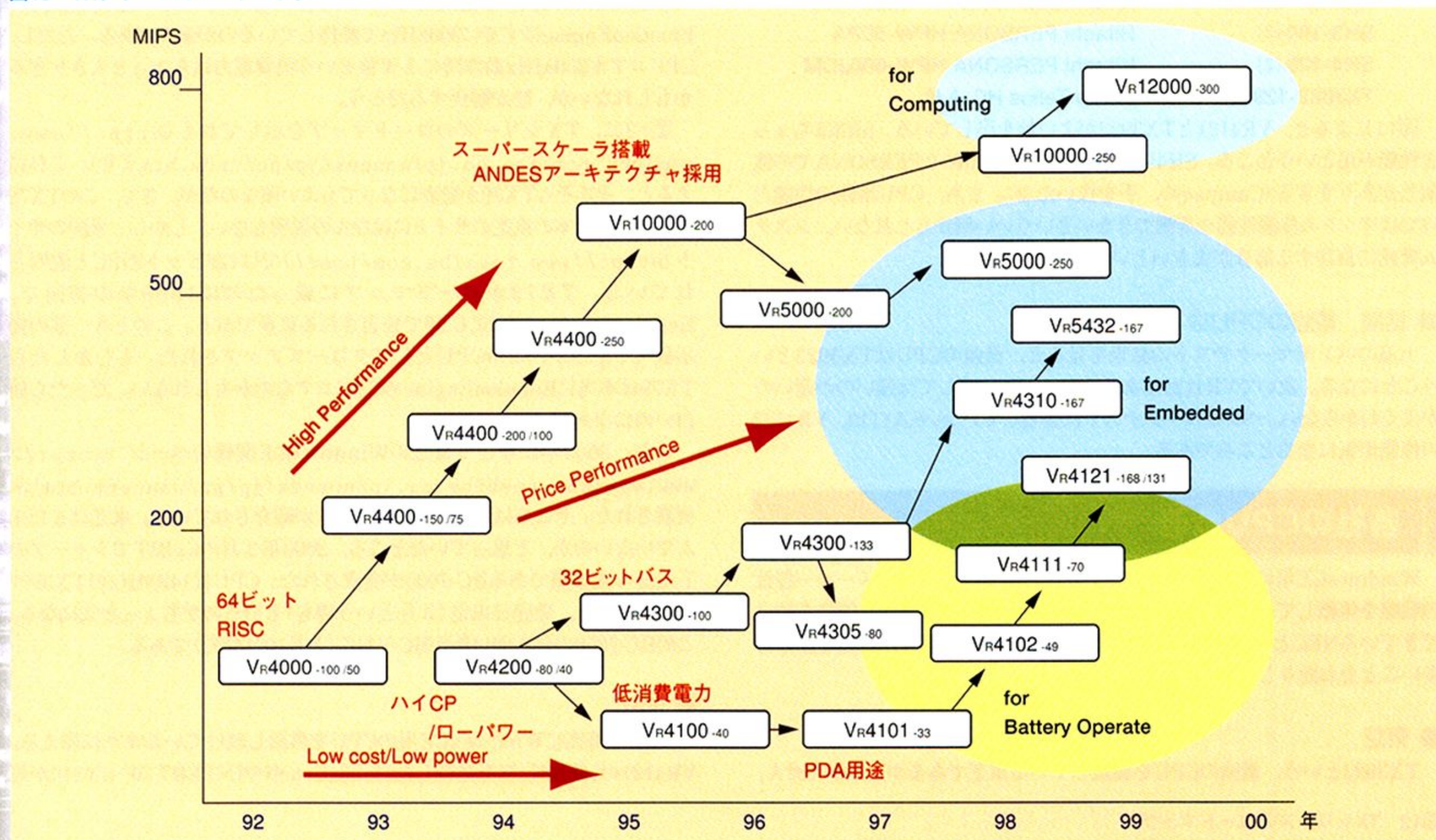
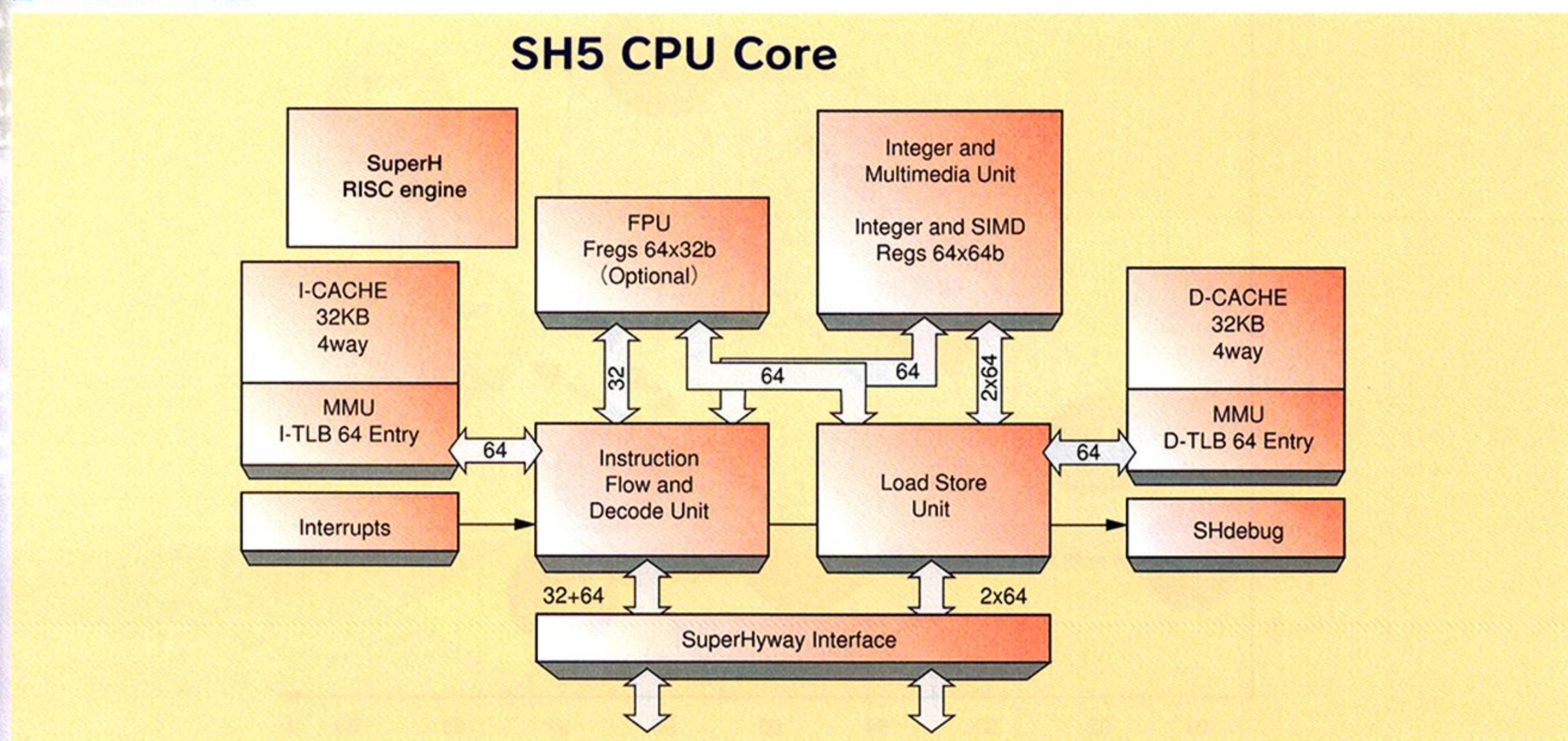



図14 SH5のブロック図

てはNECの独占供給になるということも十分考えられる。

そういえば，WindowsCE FANでVR4121と同性能で消費電力が半分というPsPC用のVR4181というCPUが紹介されていたが，これはどうなったのだろう。正式に発表されたという記憶はないのだが。2000年３月時点でのマイクロソフトの認定CPUの一覧にはVR4181の名前があるので，登場は近いかもしれない（と思っていたらRapierで採用予定とか）。

図13にVRシリーズのロードマップを示す（http://www.ic.nec.co.jp/micro/product/index_vr.htmlより）。2000年４月時点でVR4122はまだ載っていない。ただ，CASIOはRapierで採用予定のようである。MIPS系のCEマシンに関しては，東芝の将来性を不安視する声があるのも確かで，現在TX3922を採用しているメーカーがVR4122あたりにCPUをリプレースする可能性も十分ある。

● 日立

日立も着実にSH3，SH4，SH5と性能を向上させている。SH5では互換性の点でSH4の轍を踏まないでほしい。ただ，SH5では動作周波数を向上させるためにSH4のウリであったスーパースカラをやめてしまったのが残念である。SH4とSH5は同一周波数で見れば性能（MIPS値）が変わらない。SH4のスーパースカラの効率が悪かったのか，SH5で32ビット命令長を採用した効果が絶大だったのかは不明である。まあ，SH5は単に高速版SH4としての位置付けであろう。日立としては400MHz動作という宣伝文句がほしかったというのは穿ちすぎか。まあ，16ビット長命令から32ビット長命令への切り替えの契機になるCPUであることは間違いない。

それにしても，WindowsCEにおいて，400MHzという高周波数でSH5が使用可能かどうかわからない（消費電力の点で）。カタログどおり消費電力を400mWに抑えることができたとしてもちょっと多い。低消費電力を目指して特別に開発されたSH4低消費電力版は（これがWindowsCEマシンに採用されているのだが）動作周波数は128MHz止まりだった。現実的には，SH5のWindowsCEマシンでは250MHz～300MHz動作のものが使用されるのではないかと推測される。このときの消費電力は300mW程度であろうか。たとえ200MHz動作に終わってもキャッシュ容量が増えている分，同一周波数のSH4よりはシステム性能はよいはずである。また，その場合でも性能は300MIPSを超えるので，WindowsCEマシンに採用されれば最強のマシン候補になることは間違いない。

図14にSH5のブロック図を示す（Microprocessor Forum'99より）。また，SH5をCPUコアとする最初の周辺機能内蔵チップのブロック図を図15に示す。SHシリーズ全体のロードマップを図16に示す（http://www.super-h.com/virtual_expo/seminar/ess1/01_1.htmlより）。

ところで，日立はWindowsCE戦略において方針転換をしたのではないかと思われる節がある。なぜか最近SH4を採用したCEマシンが登場しない。PostPet内蔵のH/PCのCPUがSH3だったのは不思議だったが，あれはWindowsCEのバージョンが昔のものだったので，それなりには納得していた。Rapierも（現実的には）SH3版しか予定がないようだし，SH4ではコスト的に割が合わないのだろうか。SH4の動作周波数が128MHz以上に上がらない（マイクロソフトは167MHz版も認定している）のでSH3と差別化が図れないという噂もある。このSH4は，Dreamcastに採用されている電力馬鹿食いの200MHz版とは異なるので注意。

なお，日立は2000年５月11日に新しいPERSONAのラインアップを発表した。HalfVGAのHPW-60PAとVGAのHPW-650PAがそれである。CPUはそれぞれ，SH3-100MHz，SH4-128MHzである。発表時期も搭載CPUも中途半端の感は否めない。この時期にWindowsCE2.0（HPW-650PAはCE2.11）を搭載したマシンを出す意義がわからない。それはともかく，SH4は今回もVGAモデルにしか使用されないみたいだ。

● インテル

StrongARMの特徴はなんといっても低消費電力である。SA1100は190MHz動作を実現している割に，性能はお世辞にもいいとはいえない。モノクロマシン（というか１世代前のCEマシン）をターゲットにしているのではないかとも思える。性能の出ない（公式な？）理由はバス転送能力が低いということで，バス速度を向上させたSA1500という製品が計画されていたような記憶もあるが消息不明である。現在は，バス速度を103MHzに向上させたSA1110（動作周波数は206MHz）を積極的に売り込んでいるようである。新しいTeliosに採用されたTX3922の例もあるので，バス速度の高速化である程度の性能向上は見込める。しかし，本命は以下に示す次期StrongARMだという見方が強い。

StrongARMはWindowsCEではぱっとしなかったが，ここにきてインテルはついに本気になったようだ。DECによるStrongARMの設計を放棄し，自社開発に踏み切った。その性能たるや，600MHzで動作し，CPUコアの消費電力が0.1mW（１チップでは450mW）という，インテルでなかったら戯言としか受け取られない高性能である。この数値を競合他社はかなり脅威に感じているはずだ（ただ，実際の性能は600MHz動作で約700MIPSと控えめではある）。有言実行できるか否か，その答えは近いうちに出る（はずなのだが）。StrongARM陣営の巻き返しはなるか。なお，インテルは，WindowsCEマシン用としては動作周波数が400MHzのものを推奨してい

図15 SH5をCPUコアとする最初の製品

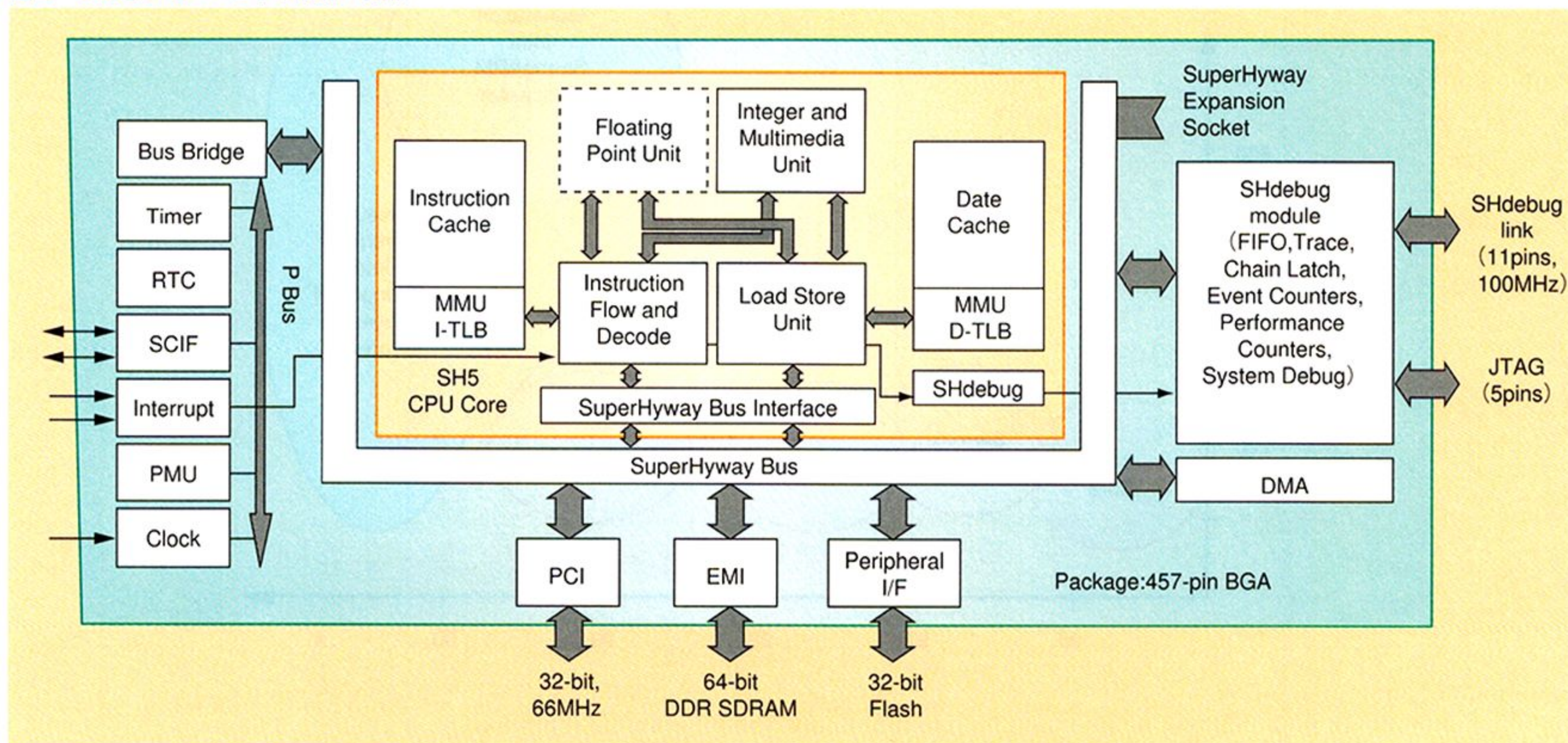



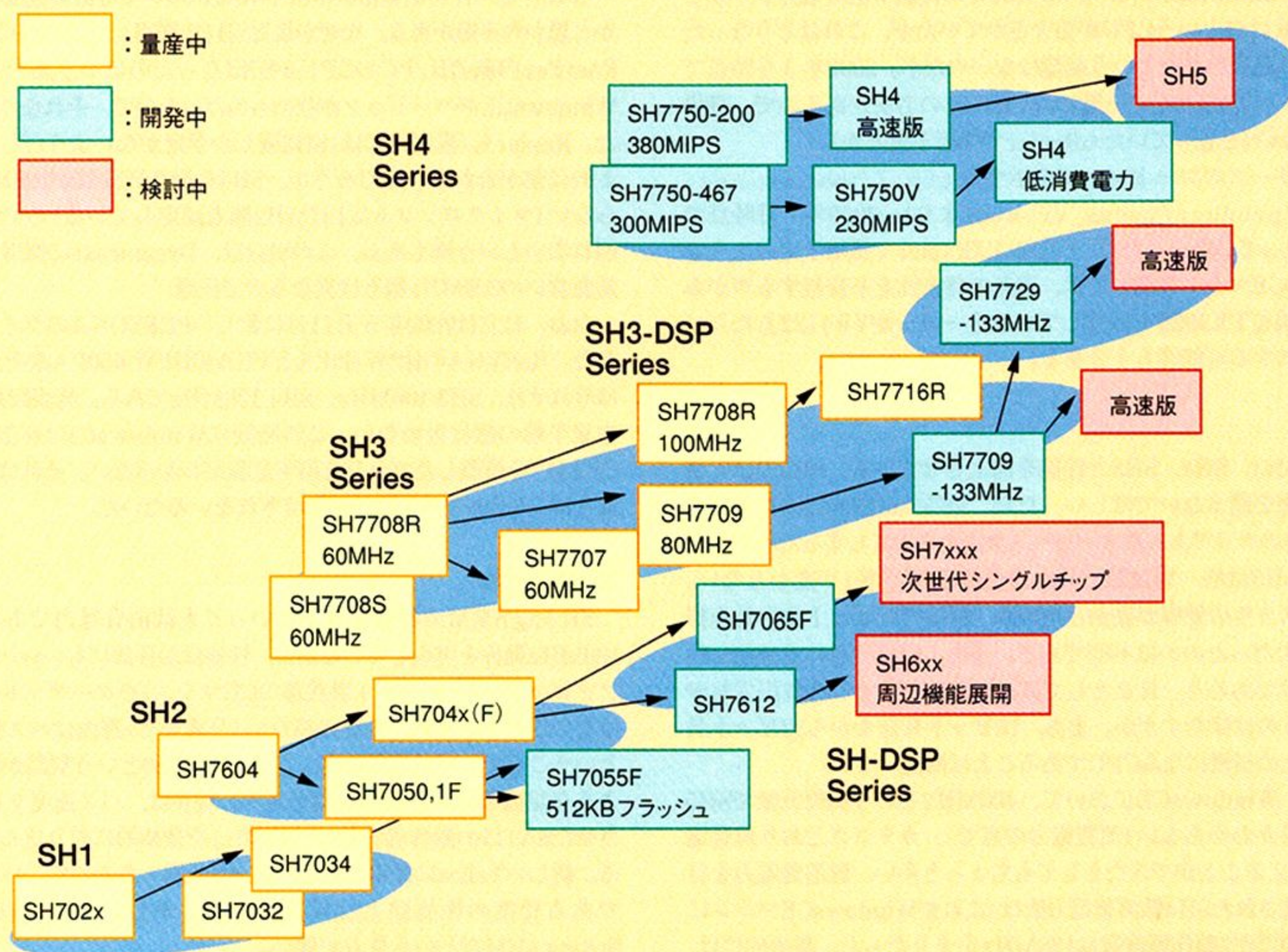
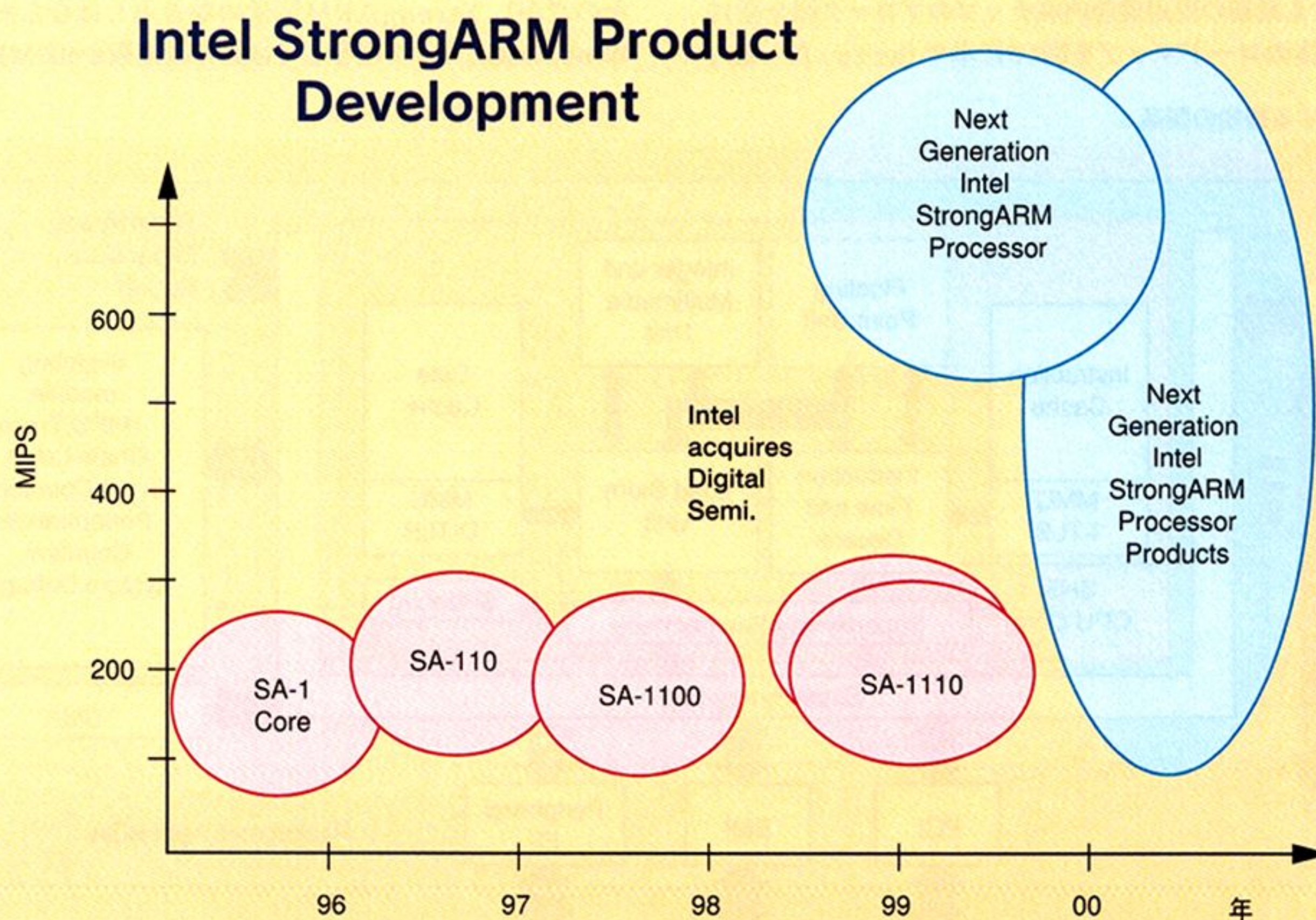
図16　SHシリーズのロードマップ

※F-ZTATTM（Flexible-ZTAT）は，㈱日立製作所の商標です。
※superHTMは，㈱日立製作所の商標です。

図17　StrongARMのロードマップ

るようである（このときの消費電力は230mW）。それでも最強のマシン候補になることは容易に推測できる。事実，次期WindowsCEマシンや組み込み機器にはこの新StrongARM（業界ではSA2と呼んでいる）が結構採用される予定だと聞いている。図17にStrongARMのロードマップを示しておく（Intel Developer Forum'99より）。

SA2に関しては2000年初めのIntel Developer Forumで発表されるとの憶測が飛んでいたが，結局はなんの発表もなかった。開発は順調に遅れているのだろうか。競合他社は胸をなでおろしていかもしれない。インテルとしては，SA1110が好評なので，SA2が遅れてもそれほどあわてていないような気がする（噂では，水面下で着々とSA2の商談が進んでいるという）。また，一部の報道によれば，カラー版Palmも将来データ通信をサポートするようになるとDragonballでは性能不足になるのでSA1110の採用が検討されているという。WindowsCEとPalmの両方でデザインインできたら無敵である。

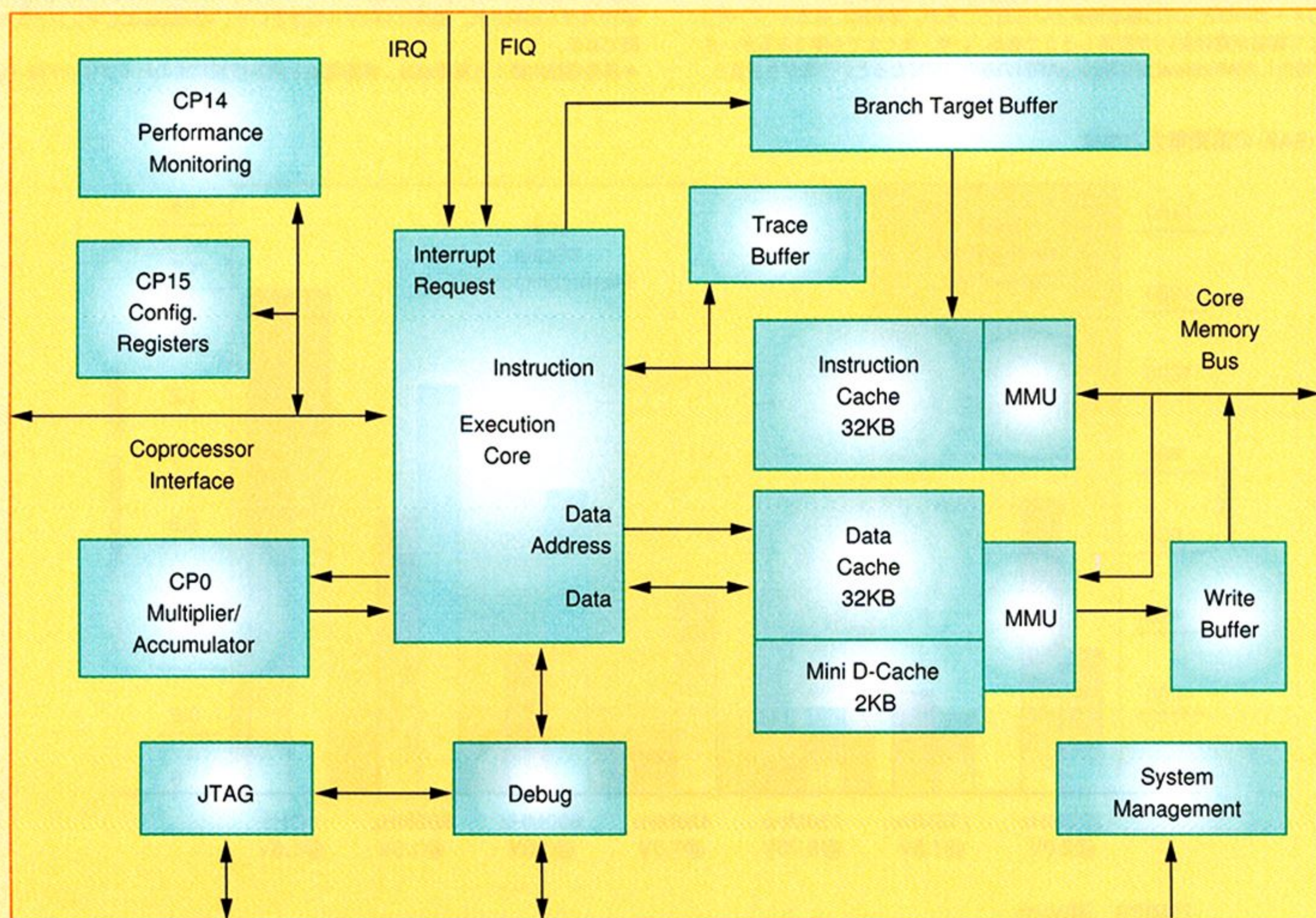
図18にSA2のブロック図，図19にXScale（SA2）の従来品との性能比較を示す。

- PDAで通信機能をサポートするのが流行であるが，ヨーロッパにおいて通信系の組み込み機器で実績のあるARMが注目を集めている。IPコアの分野でもARMは，MIPSと並び，2大勢力を形成している。GameBoy AdvanceのCPUもARMであるし，ARMアーキテクチャは今後ますます浸透していきそうだ。
- 2000年7月10日付のEETimesによると8月22日から開催されるIntel Developer ForumでSA2の発表があるらしい。当初の予想を裏切り150MHz品のみの発表ということである。インテルの話によると日程の遅れは「プロセス技術のトラブルではない」ということだが，それって「プロセス技術のトラブルで遅れました」といっているのと等価では。別の情報筋によると，今年発表されるSA2はCPUコアのみで周辺内蔵品は来年という噂である。150MHz品の電源電圧は0.75Vであるが，このような小さな電圧で周辺チップとどのように結合するのかが見ものである。インテルに好意的に考えれば，将来性がない（と思われる）H/PC分野への参入は中止し，今後爆発的な市場の拡大が予想される携帯分野にターゲットを絞り込んだと見ることができる。そこで必要となるのは超低消費電力のCPUである。ちなみにSA2の消費電力は40mW（150MHz動作時）といわれている。
- XScale：2000年8月23日，予定どおり，Intel Developer ForumにてSA2の発表があった。で，フタをあけてみると話が大きくなっていた。SA2は，「第三世代（3G）」の携帯電話機用に開発したXScaleというマイクロアーキテクチャを実装し，最高1GHzの周波数で動作するという。しかも，そのときの消費電力はわずか1.5Wというから驚きである。性能は約1300MIPSとか。1GHzで動作する試作品のデモも行われた。しかし，ターゲット市場の本命は携帯電話であることを匂わせてもいる。この場合，50MHzで動作させることになり，そのときの消費電力は10mWという。これは従来のCPUなら待機時並の低電力である。インテルによれば，単3電池1本で1週間動作し続けることが可能らしい。また，インテルは9月には日本での携帯電話戦略を発表するという。SA2がPocketPC（PsPC）やH/PCに使われる目はなくなったと見ていいだろう。XScale自体はスーパーパイプラインで動作周波数を向上させるとともに，PentiumIIIの『SpeedStep』機能に似たダイナミックな電圧管理技術によって低消費電力を実現する。また，メディア処理に適したSIMD（Single Instruction Multiple Data）命令を提供する。コード効率を高めるためのThumb命令セットもサポートするらしい。

WindowsCEの将来

COMDEX/Fall'99では前年に比べてWindowsCEの展示が減少した。これを受けてWindowsCEの終焉を予測するマスコミが多い。いわく，Compaq，ソニー，Philips，EverexなどのメーカーがWindowsCEベースの機器の開発中止を表明した。すでに8割近いシェアを持つPalmOSを逆転することはできない。マイクロソフトのビル・ゲイツは基調講演で「MSN Web Companion」という情報端末を発表した（一応，WindowsCEベースとされている）が，WindowsCEについては明言していない，といった具合である。ハンドヘルドやPDA市場が今後も拡大を続けるという予測に異を唱えるアナリストはいないので，PalmOSの市場がますます好調になるということであろう。このような状況を鑑みてか，イメージを一新するため，マイクロソフトは2000年からWindowsCEの名称をWindows Powerdに変更すると発表した（とCNETが報道した）。「力強さ」を必要以上に強調している気がしないでもない（負け惜しみ？）。

しかし，日本のWindowsCE事情を見る限りはそれほど悲観的なものではない。明らかにPalmよりもWindowsCEのほうが普及しているし（敵はZAURUSくらいか）H/PCはPostPetやATOKを標準装備し，ポケットボードを卒業した女性層をターゲットに支持層の拡大を狙っている。合法かどうかは置いといて，GameBoyエミュレータもあるのでその筋への普及も考えられる（Palm用にもあるが，到底実用的な速度ではない）。MP3プレイヤーとしての発展も考えられる。なによりも，WindowsCEはEメールやインターネット接続の新しい手段を実現した先駆者であるし，これからもこの勢いに乗っていくのではないだろうか。2000年4月に発表されたPocket PostPet

図18　SA2のブロック図

はWindowsCEをベースとしながらもPostPetの操作性のみに注力し，CEマシンであることを覆い隠している。これもひとつの方向性であろう。

おわりに

本稿を最初に書いてから10カ月が過ぎた。当時は海のものとも山のものともわからず，逆に，そのせいでむしろ将来を期待されていた感もあるWindowsCEであるが，現状はかなりトーンダウンしている感は否めない。単なるPIMのほかにEメールやWWWのブラウザとして新たな道を切り開いてきたが，ほとんどの機能は携帯電話に追いつかれてしまっている。残っている付加機能は音楽を聞くことくらいであろうか。起死回生で登場したPocketPCもGameBoy Advanceほどには将来性を感じないのは筆者だけなのか。個人的にはWindowsにも匹敵する「重い」OSをポケットサイズに押し込めようという発想に無理があるような気がする。Linux程度の比較的「軽い」OSでシステムを構築するほうがいいのではないか。実際，LinuxベースのPDAもいくつか登場してきているようだ（そのほとんどがStrongARM対応なのは面白くないが）。

しかし，携帯電話の飛躍的な進歩を見ていると，WindowsCE（やPalmなど）のPDAが持つ機能は，最終的にすべて携帯電話にインテグレートされてしまうのでは，という考えを捨てきれない。その意味で，携帯電話を主要ターゲットに変更したインテルの戦略は的を射ていると思う。そこで使用されるCPUは，より高速，より低消費電力を目指したものになっていくに違いない。生き残るCPUは，やはりARM系なのかなあ。

●H/PC 2000：なりを潜めていたH/PC用WindowsCE3.0であるが，2000年９月７日にHandheld PC 2000としてMicrosoftから発表された。仕様 はPocketPC＋αといったところである。当初は業務用ということだが，すぐにコンシューマ用も出てくると予想される。最初の製品はHPのJornada720といわれている。CPUはSA1110/206MHzという。ついにHPもSH3からStrongARMに乗り換えた模様。かつて，SA1100/190MHzを採用して思いっきり失敗した経験を忘れたのだろうか。それともInte l の独占が始まる兆候と見る べきか。一応，MicroSoftのプレスリリースでは，HP，NEC，MainStreetNetworks が製品開発を表明しているようである。NECのCPUは不明だが，MainStreetNetworksはVR4121/168MHzらしい。

●暗い（？）話題：マイクロソフトはWindowsCEのサポートを簡略化するため，サポートするCPUを１～２社に絞り込みたい意向という噂がある。となると，PCの分野でのつながりが強いインテルがもっとも有力である。マイクロソフトのゲーム機であるX-BOXでAMDに内定していたCPUが急遽インテルに変更になった事例もある。かつて，いろいろなCPUをサポートすると表明されていたWindowsNTがx86系１本に路線変更された経緯もあり，インテル（やAMD）以外のCPUメーカーにとっては悪夢再来といったところか。客観的に見ると○○や□□（クレームが怖くて実名が書けない）が脱落しそうである。いや，あくまでも噂であるが。さらに，携帯電話に特化したWindowsCEはStrongARMのみの対応になるという噂がちらほら。

●明るい（？）話題：2000年６月にCompaqのPsPCであるPRESARIO 213が１万円以下の超破格値で放出され大ブレイクしている。少し前，米国ではIBMのH/PCであるWorkPad z50が199ドルで在庫整理（?）されていたが，あまり話題にはならなかった。今回もCompaqがiPAQ H3600やAero1550の発売を控えて旧型機種を在庫整理していると見ることもできるが，さすがに１万円を切るとユーザーの受けが違うようである。モノクロ画面である点やCPUの非力さ（VR4111-70MHz）はあまり問題になっていない。この現象をメーカーはどう分析するのであろうか。

●余談：2000年７月時点では，HandSpring社のPalm互換機である『Visor』が，その低価格ゆえに爆発的ともいえる人気を博している。特にPDAの初心者の注目度が高いという。９月にはSONYのPalm互換機の発売も予定されているし，Palmの人気はこのまま続くのだろうか。同じく９月に登場が噂されているPocketPCの日本語版でWindowsCEがどのように市場に食い込んでいくかがが見ものである。

●ソニパ：2000年７月13日，SONYのPalmOS搭載機である『PEG-S500C』（カラー液晶），『PEG-S300』（モノクロ液晶）が正式に発表された。９月９日の発売という。気になるCPUはDragonball EZ 20MHzということだ。CPUだけに関していえば意外性がない。ところで，カラー版もモノクロ版も電池寿命が約15日というのは本当なのかな。価格は，オープンとなっているが，カラー版が６万円前後，モノクロ版が5.5万円前後といわれている。Palmの特徴はなによりも安いことではなかったのか。なお，８月17日，ソニーはこのPalm互換機の名称を『CLIE』とすると発表した。“Communication Linkage for Information & Entertainment”の略らしい。

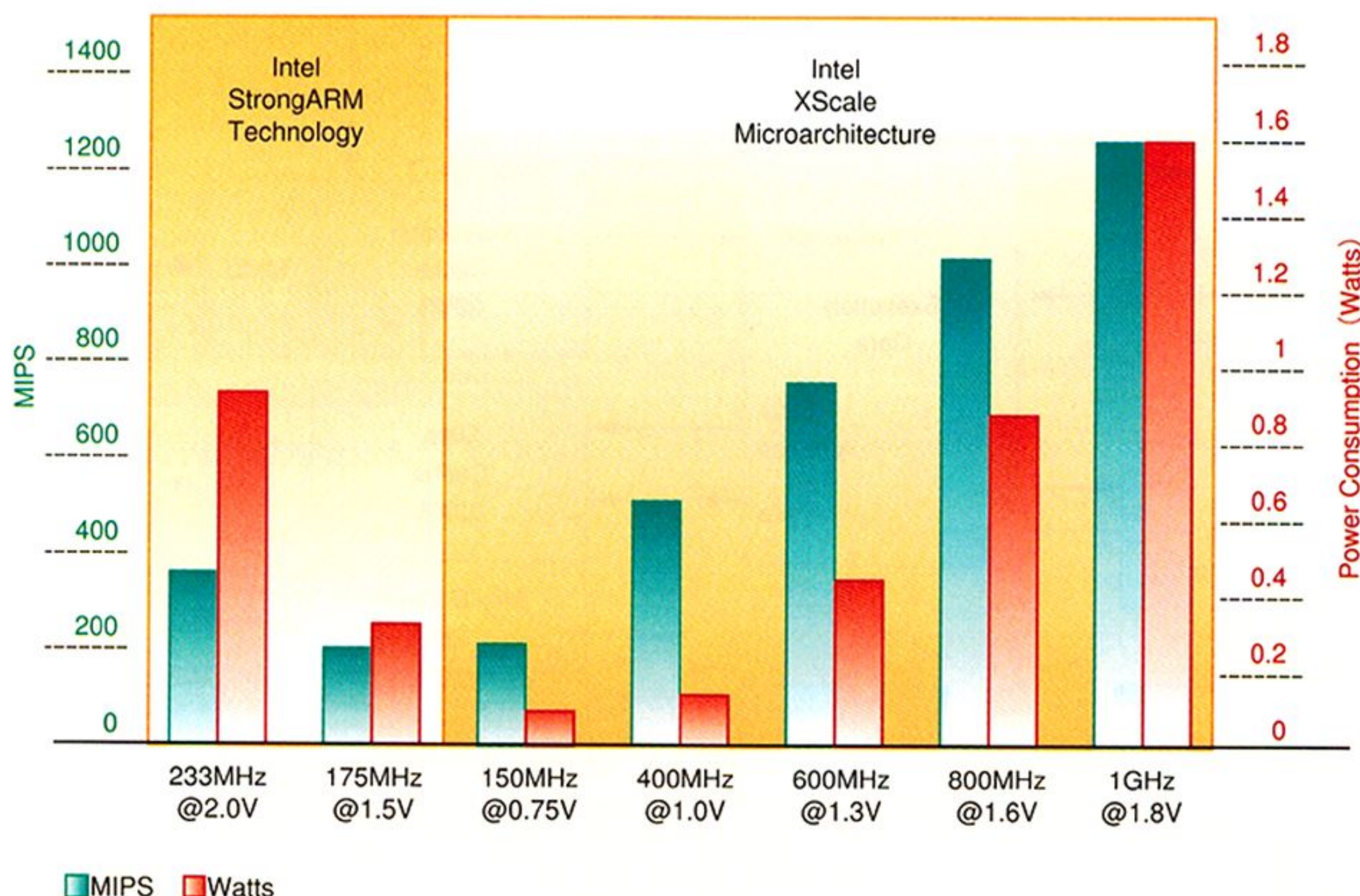
●Windows Powered：SONYのPalm互換機の発表と同じく，７月13日にはマイクロソフトがWindowsCE3.0の日本語版のOEMメーカーへの供給を始めたと発表した。いまさらながら，カシオとHPが日本語版PocketPCの発売を表明した。PocketPCの正式名称は『Microsoft Windows Powered Pocket PC』というらしい。こちらも９月発売予定だとか。ただ，Compaqが発売を表明していないのが気になる。CPUであるStrongARMの供給に難があるのだろうか。米国であれほど評判がよかったiPAQの日本語版を出さないはずはないと思うのだが。StrongARMの方針転換（携帯電話へ特化）が影響しているのだろうか。

●日本語PocketPC第１号？：2000年８月９日，カシオはPocketPCをベースにした業務用高性能ハンディターミナル『CASSIOPEIA DT-5000』を11月10日より発売すると発表した。CPUはVR4122の150MHz品である。2000年８月21日にはPocketPC日本語版である『CASSIOPEIA E-700』が発表された。CPUは同じく，VR4122の150MHz品である。発売は９月15日。価格はオープンとしながらも，６万円前後と予想されている。

●HP製PocketPC：2000年８月16日，日本HPはPocketPC日本語版を搭載した『HP Jornada 548』を10月５日より発売すると発表した。CPUは予想どおりSH3-133MHzである。販売開始1年間での目標販売台数は５万台。価格はオープンであるが，約６万円とか。少し高い。

●時代はLinuxへ？：Linuxを搭載したPDAの発表が相次いでいる。2000年８月14日から開催されたLinux World ExpoでCompaqは『iPAQ H3600』上にLinuxを移植したPDAを発表した。WindowsCEのフラッシュROMをLinuxで上書きするという荒業をやっている。『iPAQ H3600』は，通常のマスクROMではなく，ページモードを持つフラッシュROMを採用しているので，これが可能になる。CPUはもちろんSA1110である。iPAQ用Linuxはhttp://www.handhelds.org/downloads.htmlより無料でダウンロードできる。同時期，Agenda Computingは，Linux搭載PDAである『Agenda VR3』を発表した。筐体はWindowsCEマシンの流用ではなく専用設計らしい。NEC製のLinuxを載せるということ，動作周波数が66MHzのMIPSプロセッサであることを考慮すると，CPUはVR4181と思われる。と思っていたら，MIPSの日本語サイトでVR4181と明記されていた。『Agenda VR3』は今年10月から出荷され，価格は149ドル程度という。WindowsCEマシンと比べるとかなり安価である。

●携帯電話内蔵：三菱電機は，携帯電話と携帯情報端末のハイブリッド機『Mondo』を，9

図19　XScale（SA2）の消費電力と性能

月より欧州で発売する。WindowsCEをOSに採用した、通信機能を持つ携帯端末である。形状はPalmによく似ている。携帯電話用のWindowsCEはまだ発表されていないが、新しいOSなのだろうか。欧州ということから見て、CPUはARMまたはStrongARMと見て間違いないと思うが。

●日本語版「Handheld PC 2000」：2000年10月10日、MicroSoftはキーボード付きPDAの規格「Microsoft Widows Powered Handheld PC 2000」の日本語版を発表した。WindowsCE 3.0をOSとする。H/PC 2000を搭載するPDAはシャープ、NEC、JVC、HPが年内に、富士通が来年以降に発売する予定。ところで、H/PC 2000がサポートするCPUはMIPSとStrongARMのみというアナウンスがなされている。WindowsCE普及の功労者であるSHを切り捨てたところに意味深なものを感じる。

●MC/R550/R450：日本語版H/PC 2000の発表を受けて、10月11日、NECはそれをOSとする新型モバイルギアIIを発表した。CPUはVR4121-168MHzで前機種と変わらない。価格はR550が98000円と同じCPUを採用している（TVCMで有名な）sigmarion（OSはWindowsCE2.11で本体はNECのOEM）の59800円より遥かに高い。sigmarionのWindowsCE3.0版だったらと残念がる声は多い。なぜCPUが最新のVR4122でないのかは不明。

●HC-AJ3/HC-VJ2C：10月13日、シャープも日本語H/PC2000を搭載する新型Teliosを2種類発表した。従来のHC-AJ2/HC-VJC1の後継機種である。CPUはMIPSRISCとしか公表されていない。ただ、ハードウエア仕様が従来機とほとんど変更がないので、CPUもTX3922-148MHzのままと予想される。価格はオープンだが、HC-AJ3が13万円前後、HC-VJ2Cが14万円前後と予想されている。強気だ。

●MP-C303：2000年10月17日、JVCからの正式発表がないまま、WORLD PC EXPO 2000でH/PC 2000をOSとする新型InterLinkが動体展示された。AV機器を念頭に置いた、その売りは16万色同時発色の1024×600ドットのワイドSVGA（でも7インチと小さい）と音源のステレオ再生である。CPUは「現時点では公表できない」という話だったが、新MobileGearIIや新Teliosよりも数段機敏な動作をしているところ見ると、VR4122-180MHzではないかと推測される。11月1日に新型InterLinkはJVCから発表され、その後、CPUはVR-4122/180MHzであることが発表された。

●Jornada720日本語版：年内発売予定のH/PC 2000を搭載したPDAはWORLD PC EXPO 2000で発表されるとの報道があったが、日本HPからはなんの発表もない。おそらくはJornada720の日本語版が発売と思われる。その場合、CPUがSA1110-206MHzなのは既報のとおり。ただ、iPAQ H3600の場合と異なり、バスクロックが51MHzとの噂がある。バス速度が性能に影響するWindowsCEにあってはあまり性能に期待ができない。Jornadaファンの期待を裏切ることがなければよいが。11月9日に正式発表、12月に発売された。

●PCN-C600：WORLD PC EXPO 2000で発表されたH/PC 2000マシンで異彩を放っていたのは高木産業が発表したCeleron/600MHzをCPUとするA4サイズのノートPCである。気楽に電源をON/OFFでき、電源ONですぐ起動をポリシーとしている。さすがに長時間のバッテリ駆動は不可能と思われるが、PDAとは違う発想であると割り切っているところが凄い。体感速度は、H/PC 2000マシンのなかでは抜群に速い。さすが600MHz動作。

●SH6/SH7：2000年10月12日、日立とSTMicroはSH5に続き、SH6、SH7も共同開発することを発表した。SH6の動作周波数は1GHz以上。SH5のマイクロアーキテクチャをベースとし、ネットワーク機能を強化する。2002年後半発売予定。SH7ではマイクロアーキテクチャを見直す。詳細は未定。

●TX79続報：Microprocessor Forum 2000で発表されたはずの128ビットRISCのTX79であるが、反響がまったくない。MIPS社のプレスリリースでも「TX79はネットワークとデジタル家電を応用分野とする強力なIPコアになる」という程度の説明しか載っていない。EmotionEngineのCPUコアという予想（妄想？）は当たったのか外れたのか……。もしかしたら、発表はキャンセルされた？

●新DragonBall：2000年12月、米国Palm社はPalmOS5.0ではCPUとしてARMを採用することを表明した。さまざまなARMアーキテクチャと互換性を取るため命令セットはARM4T（ただしThumbは使用しない）を使用する。従来のDragonBallにはエミュレーションで対応する。なお、モトローラがARMとライセンス契約を結び、新型DragonBallを開発中との噂もある。

■追記1

Winter CES 2000において，「MSN Web Companion」が実際に発表された。これはWindowsCE2.12上でフルセットのIE4を動作させるものらしい。CPUはx86ということである。MSNというのはMicrosoft Networkのことで，マイクロソフト独自のネットワークサービス(というかプロバイダサービスといったほうがよいかなあ)のことである。また，同時に噂の「Rapier」も発表され，それを搭載するマシンは従来の「Palm-size PC」から「Pocket PC」に名称変更になったようだ(バージョンはついにWindows CE3.0になったという)。マイクロソフトの製品がまともになるのはバージョン3以降という定説は今回も当てはまるのだろうか。ともあれ，WindowsCEはまだまだ健在といったところか。それにしても，WindowsPowerdという言葉はまったく定着していない。名称変更の発表自体がガセだったのかもしれない。いまではCNETの誤報というのが定説である。まあ，それほどまでにWindowsCEが(圧倒的シェアを誇るPalmの対抗馬として)注目に値する製品というのは間違いないのだろう。

●2000年7月4日，マイクロソフトの日本法人はMSNのプロバイダ事業からの撤退を表明した。「MSN Web Companion」の運命は？　それに対し，2000年8月15日，MicrosoftとCompaqは，MSN専用端末「MSN Companion」シリーズの初の製品となる「iPAQ Home Internet Appliance」を発表した。価格は599ドルだが400ドルのキャッシュバックがあるので実質199ドルである。日本での発売は，もちろん(?)，未定。いや，もしかして，「ぷらら・ウェブ・コンパニオン」だったりして。

■追記2

と思っていたら，ZDNetによると，最近(2000年2月)WindowsCEマシンの売り上げが芳しくないらしい。CEマシンはまだまだ高価で，実際に売れている携帯情報端末は，コミュニケーションパル，ポケットボード，ザウルスアイゲッティなどの安価で操作が簡単なものに人気が集まっているらしい。簡単にメール，インターネット接続をウリにしたキーボード付きCEマシンは完全に後れをとっている。かろうじて，MobileGearのHalf VGAモデルが健闘している程度とか。

ビジネスマンをターゲットに，鳴りもの入りで登場したSVGA/VGAモデルはこぞって売れ行き不振で値下げをやむなくされている(そこそこ売れているTeliosだけは強気だそうだが)。この先，WindowsCEマシンの生き残る道は従来の機能をいかに安価に提供できるかにかかっているといっても過言ではないだろう。とはいえ，ZDNetでは新型Telios(HC-VJ1C)とJornada690の在庫状況が頻繁に報告されている。ユーザーの注目はこの2機種に集まっていると見られている。最近(2000年4月)の報告ではHC-VJ1Cが呼び水となってほかのCEマシンの売り上げも増進しているらしい。まさに水ものである。H/PCの明日はどっちだ。

それにしても，PsPCのほうはどうなのだろう。現状はカシオの独占状態であり，Rapier搭載のCASSIOPEIA(E-115という機種が4月19日に発表された)はカラー版PalmPilotといい戦いをするとは思う。しかし，個人的には携帯電話やGameBoy Advanceがメールやインターネット接続のインフラを整備したら，それにはかなわないような気がする。最近(2000年4月)，部品の調達不足を理由にGameBoy Advanceの発売が延期されてしまった。発売延期は任天堂の常套手段(?)なので驚きはしないが，少し残念である。

●GameBoy Advance：8月24日の任天堂の発表によると，発売日は2001年3月21日，価格は9800円とのこと。携帯電話との連携がナウい（死語）。たしかCPUはARMだったはず。周波数は不明。サブCPUとして8ビットCISCの存在も公表されている。こちらはZ80か。同時にNintendo Game CUBE（コードネームDolphin）も紹介されたようである。2001年7月発売予定とか。価格は未定のようである。本題とは関係ないが，Game CUBEのCPUはPower PC "Gekko" 0405MHzだそうである（PPC750のカスタム）。整数性能は925MIPS。噂の750MHzではなかった。でもL2キャッシュは内蔵だった。ところで，VRAMが全部で3MBしかないけど大丈夫か？　（編注：VRAM内にはバックバッファ，Zバッファ，テクスチャバッファが取られ，フロントバッファはメインメモリ側に取られる構成という。描画用に2MB，テクスチャ用に1MBということなのでPS2よりは遥かに多く，しかも容量から見てノンインタレースを前提にしていることがわかる。問題はメインメモリへの転送コストだが……）

■追記3

2000年4月19日，各社のPocketPCのラインアップが一斉に発表された(表6)。CPUの採用状況を見るとNECのVRシリーズが抜きん出ているようである。しかし，仕様的な魅力はCompaqのStrongARMマシンだろう。聞くところによると，SA1110の評判は非常によく，ほとんどのCEマシンのメーカーが注目しているようだ。そのウリは103MHzという高速なバス速度である。そのためか，SA1110は他社製CPUと比べて素晴らしい性能を発揮しているという。危うし，NEC，日立(東芝も？)。WindowsCEの世界もインテル一色になってしまってはつまらないと思う今日この頃。CE界のAMDになれるのはNECなのかなあ。

PocketPCに遅れること約1週間の2000年4月25日，WinHEC2000でマイクロソフトはPocketPCのOSであるWindowsCE3.0を6月に発売することを正式発表した。サポートされるCPUは表7のとおりであるという。ただし，最初の版ではARM(Thumbなし)，VR41xx(MIPS16)，SH3，x86の4種類のみのサポートと聞いている。(実際に製品に組み込まれている)StrongARMがないのが解せないところである。まあ，ARM互換だから問題はないのだが。また，マイクロソフトのWindowsCE関係のWebサイト(`http://www.microsoft.com/windows/embedded/ce/default.asp`)もPocketPC(WindowsCE3.0)中心に一新された。ここを見ると，対抗はPalmでH/PCは二の次という感がする。マイクロソフトはH/PCに注力するのをやめてしまったのだろうか。まあRapierをH/PCに転用するという方向は十分考えられるので期待だけはしておこう。

表6　Pocket PCのラインアップ

メーカー	マシン名	CPU	
Compaq	iPAQ H3600	SA1110	206MHz
	Aero1550	VR4111	70MHz
HP	Jornada 540/545/548	SH3	133MHz
CASIO	CASSIOPEIA E-115	VR4121	131MHz
	CASSIOPEIA EM-500	VR4122	150MHz
	CASSIOPEIA EG-80	???	
	CASSIOPEIA EG-800	???	
Symbol	PPT 2700 Series	VR4181	66MHz

表7　WindowsCE3.0がサポートするCPU

マイクロプロセッサファミリ	サポートされるマイクロプロセッサ
ARM	ARMx20T（Thumbも含む）
MIPS	MIPS39xx, MIPS41xx, MIPS4300（MIPS16も含む）
SHx	Hitachi SH3, SH4
PowerPC	PPC403GC, PPC8xx
x86	486, 586, Pentium, MMX Pentium, Pentium II

携帯ゲーム機/PDAを作る 第1回 プラットフォームの製作

高尾 克彦　Takao Katsuhiko

PDAというのはもうひとつオープンさに欠けるという方のための番外編です。ここではユーザーレベルでPDAを製作するのに必要な手順とその過程を追ってみました。オリジナルPDA製作の道はなかなか険しそうですが，不可能というわけではありません。

休刊前からそうだったのだが，Oh!Xの特集はいきなり決まる。編集会議が開かれたことはない。（U）氏の頭の中に突然閃き，ライター陣は次号の予告を見て「ネットワークプログラミングってなんですか〜？」とか「2Dグラフィックってなにをするんですか〜？」と動き出す。「編集さんの特集は○○かもしれませんが，私の特集は△△です」といって，第2特集をほぼひとりか2人で書き上げてくるライターもいる。

たしか，去年の10月，Oh!X復刊3号の打ち上げの最中だった。じゃあ，次は携帯ゲームを特集しよう，と「ゲームボーイは〜」「ワンダースワンは〜」と盛り上がっていたのだが，重大なことに気がつく。「秘密保持契約」を結ばないと開発情報が入手できない。しかも，「雑誌に発表します」という目的では，秘密保持の契約など結びようがない。

（U）氏がつぶやく。「では，作りますか」

かくして，私がCPUおよび基本ソフト，（Ussy）氏がLCD周り，菊池氏が筐体関連，（で）＆飯田氏がアプリケーションプログラム，（U）氏がマネージメント全体および契約関係を担当することとなった。

（編注：多少誤解もあるような気がするがとりあえずノーコメント）

1. CPUを選ぶ

システムを作る前に，なにはともあれ心臓部であるCPUを決めなければなりません。携帯ゲーム機ですので，とりあえず「電池で駆動する」という前提の下，低消費電力であることが大きなファクタとなりますので，PentiumIIIとかAthlonなどといったファン冷却が必要な大食いデバイスは除外します。たとえば，私の知る限り（少し古いデータですが），単3のリチウム電池の容量が1800〔mAh〕です。これは1.5〔V〕を180〔mA〕×10〔hour〕の使用ができますよ，ということなのですが，たとえば，PentiumIII（Coppermine）の消費電力28〔W〕というのは，コア部分が1.7〔V〕×16〔A〕ですから，

1.5〔V〕÷1.7〔V〕×1800〔mAh〕÷16000〔mAh〕

で，0.099〔hour〕（約6分）しか駆動できないということです。実際には440MXなどの周辺デバイスもそれなりに電力を消費しますから，システム全体では3分ももたないでしょう。Macintoshに使われているPower PCも同じ範疇です。

PCに使われているCPUを除外するとなにがあるでしょうか？　以下に思いつくまま列挙します。

8ビット系

Z80系：川崎製鉄からオブジェクトコンパチブルかつ同一クロックで動作速度が4倍というKL5C8012が発表され，再び盛り上がっています。また，Oh!MZ/X時代に培ったテクニックもまだまだ有効でしょう。必要そうなペリフェラル（PIOやSIOなど）も内蔵されていて使いやすそうです。

メモリを内蔵していないのと，Z80コアがC言語などの高級言語との相性がそんなに高くないので今回は見送りました。

メモリが外付けになるということは，第一に配線が面倒です。というのは冗談ですが，最近のようにプロセッサ自身のクロックが上がり，1命令あたりに必要なサイクル数が上がってくるとメモリに対するアクセス速度要求もその分上がってきます。CPUとメモリが別チップになっていると，CPUコア→（マージン分を差し引いたスペック規定されている）バスドライバー→配線→（マージン分過大な要求をする）バスバッファ→メモリ，と2カ所でマージンを取りあい，さらにアクセススピードの要求がきつくなります。

また，最近はプロセステクノロジの改良のおかげでCPUコア自身の消費電力は徐々に下がっているのですが，外部バスの消費電力は一定のままです（外部バスの消費電力を下げると，駆動能力も下がってしまうので）。以前は問題にならなかったのですが，最近は消費電力を下げるという観点からも，メモリの内蔵は有効です。そういえば，三菱電機製マイコンはDRAMを内蔵したらシステムの消費電力が1/3になるという売り文句でしたね（eRAMテクノロジー）。

そういうわけで，内蔵メモリ付きというのはわりと重要な要素になってきます。

PICマイコン：3号，4号でも記事になったように旬のデバイスです。ただし，小規模の組み込み用に特化しており（たとえば，アドレスレジスタも8ビットなので256バイトを越える領域はバンク切り替えを使用しなければならない。分岐命令も8ビット境界をまたぐときは，結構面倒くさい），携帯ゲーム機のプラットフォームとしては少し力不足な感じがします。消費電力や入手性は問題ないのですが……。同じ理由で，SVRやSXマイコンも除外します。

16ビット/32ビット系

H8系：秋月電子のAKI-H8の発売以来，再注目されるようになりました。特にH8/300Hファミリはメモリ空間も16Mバイト，拡張レジスタ幅も32ビットと強力になりました。また，H8/3048はプログラムROMの代わりに100回まで書き換え可能な128Kバイトのフラッシュメモリを搭載してします。また，ノートPCのスタンバイ時のシステム管理などに使用されていたりと，低消費電力にもそれなりの実績があります。

SH系：上記H8の後継CPUとして開発されました。SEGA SATURNに採用されたSH2，Dreamcastに採用されたSH4が有名です。サードパーティ製ボードや開発ツールなどが手頃な価格で出回っているので（それでもひととおり揃えると，10万円中ぐらいになる），採用することも不可能ではなかったのですが，とりあえずH8で試して，ダメだったらSHも検討するというスタンスで行くことにし，今回の採用は見送りました。

ARM：個人的にはいちばん注目しているのですが，我々アマチュアにとって入手製に問題があります。ARM社は直接デバイスを作らず，半導体メーカー（シャープなど）にデザインを販売するという会社です。半導体メーカーは，ARMコアにさまざまな周辺回路を集積しデバイスとします。このようなデバイスは汎用品というよりは，特定のアプリケーションの特定のモデルに特化した製品となっており，一般に我々アマチュアが入手することは非常に困難です。

実際には，AppleのNewton（ARM4?）やシャープのザウルスに使われていました。

MIPS：ARMと似た状況にあります。注目してはいるのですが，入手は困難です。PlayStationに採用されたCPUとして有名ですね。Philips社のVelo 1というWindow CEマシンに採用されていました。第1世代の

Windows CEマシンは主にSH2/3を使用していましたが，例外的にVelo 1はR3910という，1世代前のデバイスを採用していました。新しいSH2/3（クロックは40～45〔MHz〕のようです）のほうが，Windows CEに特化したデザインを行っているはずなのですが，結局，第1世代のWindows CEマシン間ではVelo 1（ちなみにクロックは，約37〔MHz〕）がいちばん性能がよかったという実績があります（編注：Velo1はメモリにDRAMを採用していた。WindowsCEの場合，キャッシュ性能，メモリ性能でほとんど勝負が決まる）。

以前は，RISCプロセッサの常としてコード密度が低かったのですが（同じプログラムでもRISC用に書かれたプログラムのほうがCISC用に比べてサイズが大きくなる），MIPS16というファミリからはARMのThumbと似たような手法によって改良が加えられ，携帯ゲーム機のプラットフォームにより適したデバイスとなっています。

Cold Fire：モトローラの68000を改造し，RISCプロセッサとして復活されたデバイスです。複数サイクルを必要とする命令は削除されてしまいましたが，主立った命令はそのまま継続されています。よって，我々のX68000ファミリで培ったテクニックがかなり活用できるはずです。また，数は力の理論で，x86系についで，2番目に多くの組み込み用途で採用されているデバイスファミリですから，コンパイラメーカーもそれなりに力を入れています。プロセッサの性能差の3割くらいは，コンパイラのデキによって左右されますから，よいコンパイラがあるということは重要なことです。

プログラムメモリ

先ほど少しお話ししましたが，電池駆動を前提とするとどうしても組み込み用CPUのなかから選択しなくてはならないことになります。これらの組み込み用CPUは実装面積を減らすため，部品点数を下げるため，消費電力を下げるため，とさまざまな理由でメモリを内蔵しています。データメモリのRAMを内蔵する分には，こちらも歓迎なのですが，プログラム用メモリを内蔵されるとちょっと困ります。デバイス中に内蔵されているということは，パッケージング工程の前にプログラムを書き込まなければならないということです。これらは半導体の工場で焼いてもらうのですが，

一度焼き込んだプログラムは変更できない

プログラムを工場に提出してからデバイスが入手できるまでそれなりに時間がかかる

そしてなんといっても，

ウェハーパターンに焼き込むのでマスクパターン代がかかる。そのマスクパターンで作られた半導体は，最低でも1ロットすべてを買い取らなければならない

という問題があります。

たとえば，単価500円のデバイス，1ロット100個で，マスクパターン代が100万円だとひとつあたり，

1000000円÷100個＋500円÷100個＝10500円/個

というものすごく高価なデバイスとなってしまいます（しかも，内蔵プログラムにバグが発覚し，プログラム焼き直しになると二度と立ち直れない）。

これを10000個さばける自信があれば，

1000000円÷10000個＋500円÷1000個＝600円/個

となって問題ないのですが，試しに作ってみようという今回のような企画では少し敷居が高すぎます。

解決案として，内蔵プログラムメモリの代わりに，紫外線消去可能なEPROMを内蔵したデバイスもあります。従来のEPROMなどと同様にパッケージ表面にプログラム消去用の紫外線窓が開いていて，EPROMが入っているのだとひと目でわかるマイコンです。ただし，これらの欠点として（というかEPROMそのものの欠点なのですが），書き込みに特殊な装置が必要でデバイス単価が高い（ガラス窓がついているので，とよく説明される），アクセス速度が遅い，多くは5〔V〕動作のみで，3.3〔v〕では動作しない，などといった欠点があります。

最近はこれらのEPROMの欠点を克服したフラッシュメモリを内蔵し，ユーザーサイドで自由にプログラミングを行えるデバイスが出てきましたので，アマチュアとしてはこのような機能を持ったCPUを選ぶべきでしょう（前回，紹介したPICマイコンや，今回のH8/3048Fが相当します）。

実装性

電池で10時間以上稼働することを一応念頭に置いているので，放熱が必要なデバイスは最初から除外します。となると，次に機械的に問題となるのは，デバイスのパッケージです。ハンダづけを考えると，DIPタイプのものがあるとありがたいのですが，64ピンのDIP（例：初代MC68000）は，ポケットボードの1/3くらいの大きさがあったりして，携帯性という観点からはあまり望ましくありません。正確には覚えていませんが，手に持った感じも重かったので，小さくて軽いパッケージオプションがあるデバイスのほうがよいですね。

今回は，第3回目までに企画が面白くなってきたらプリント基板を起こしてもいいよん，という編集部のバックアップがありますので，

開発時：DIPパッケージあるいは，すでにデバイスが半田付けされた評価基板

量産時（というか連載終了時）：プリント基板を起こし専門業者に0.5〔mm〕ピッチのデバイスもハンダづけも依頼する

という線で考えましょう。業界には0.5〔mm〕ピッチのデバイスを素手でハンダづけできる方もいらっしゃるようで，とりあえずQFPタイプ（板チョコみたいなパッケージに四方からピンが出ているもの）でもOKとしましょう。

最近ではi810やPCのビデオチップのように，四方からピンが出ていないようなタイプのデバイス（BGA：Ball Grid Array）もちらほら出てきました。これらのデバイスは半田玉をデバイスと基板の間に並べておいて（あるいはデバイス下部に接着しておいて），触れずに半田を溶かすという装置が必要です。我々がお願いできるようなプリント基板屋さんではなかなか扱ってくれないので，これらのパッケージしかないデバイスも除外します。

入手性

と，いろいろ述べましたが，アマチュアにとって最後はやはり入手性でしょう。いくら最高のデバイスを見つけたとしても，半導体メーカーが我々に商品を売ってくれないと話になりません。最近はRSコンポーネンツ（株）（http://rswww.co.jp）などの通販業者から部品を購入できるので，秋葉原のパーツ屋さんで売っていないからといってそれほど悲観することはありません。

また，デバイス単体で入手できるより，せめてメモリとシリアルポートくらい付属しているボードがあれば，すぐに作業を開始できます。初めて使うCPUで手配線でボードを作成してまったく動かなかった場合，使い方が悪いのか配線が悪いのかすら特定できません。市販のボードを買ってくれば，後者の可能性を否定できるので，安心してマニュアルに集中できます。また，このようなボードにはひととおりのCPUの機能（というか内蔵ペリフェラルの機能）を用いたサンプルプログラムが付いてくることも多く，新しいデバイスではなにかと参考になります。

そんなわけで，デバイス単体の入手製にも問題ない評価ボードという点で，秋月電子通商製AKI-H8（3800円！）で，断然優位なH8/3048を使用することにしました。

ツール

CPUでなにかを行う際，プログラムを作成します。プログラムを作成するためには開発ツールが必要です。Z80や68000なら，フリーソフトがごろごろ転がっているのですが，マイナーなものになるとデタラメな値段の付いたメーカー純正のツールしかなくなってしまう場合が結構あります。

最近はGNUプロジェクトがいろいろなCPU用にGCCをリリースしていたりして，PC上にUNIX環境を構築できれば，GNUでサポートしているCPUに関してはPC上で開発を行えないこともありません。

今回のH8はそこまでしなくても，AKI-H8にアセンブラ，リンカ（ともにDOS Box上で動く）が含まれていますので，Windowsがインストールされたので1台あればOKです。また，秋月電子から一部機能制限のある（アセンブラ出力できないなど）Cコンパイラが2000円で発売されています。

出力ファイルが，なぜか，Cコンパイラと同じディレクトリにしか出力できない

とか，

インクルードファイルのディレクトリ設定がなぜか失敗する

など「？」な点も多いのですが，騙し騙し使えば，使える範囲です。ユーザーインタフェイスの部分は疑問だらけですが，Cコンパイラ本体の性能/最適化はわりと優秀なようです。

さらに，日立製作所のホームページにはモニタプログラムも用意されていて，サンプルソフト規約に同意すれば無料で入手できるようになっています(http://www.hitachi.co.jp/Sicd/Japanese/Seminar/down.htm)。このモニタは秀逸で，メモリやレジスタのダンプ/更新のみならず，1行アセンブル，逆アセンブル出力，シングルステップ実行，ブレークポイントの設定，などひととおりのことが行えるようになっています。できないことといったらソースコードデバッグくらいしか思い当たりません。これらの処理をすべてH8/3048側で行っているにも関わらず，プログラムはコンパクトです。また，モニタプログラム自身のソースコードも公開されており，不必要な機能を削ってさらにコンパクトにしたり，ターミナルとの通信用のサブルーチンをユーザープログラムから共有したりと，実際の開発にかなり役立ちます。コマンドのヒストリ機能を外して再ビルドした結果，使用メモリはROM約18Kバイト，RAM約700バイトとなりました。

H8とは，どういうマイコンか

以上のような経緯で，インストラクションも知らずに選んでしまったH8/3048ですが，使ってみるとわりと素直に使えます。

表1-1に示すようなMC68000そっくりのインストラクションを持っているので，X68000でアセンブリプログラミングを行っていた方なら問題ないでしょう。

ただし，レジスタの扱いなどで細かい違いがあるので注意が必要です。以下にポイントを挙げていきましょう。

表1-1　H8/300Hのニーモニック表

機能	命令	アドレッシングモード												
		#xx	Rn	@ERn	@(d:16,ERn)	@(d:24,ERn)	@ERn+/@−ERn	@aa:8	@aa:16	@aa:24	@(d:8,PC)	@(d:16,PC)	@@aa:8	—
データ転送命令	MOV	BWL	BWL	BWL	BWL	BWL	BWL	B	BWL	BWL	—	—	—	—
	POP，PUSH	—	—	—	—	—	—	—	—	—	—	—	—	WL
	MOVFPE*，MOVTPE*	—	—	—	—	—	—	—	B	—	—	—	—	—
算術演算命令	ADD，CMP	BWL	BWL	—	—	—	—	—	—	—	—	—	—	—
	SUB	WL	BWL	—	—	—	—	—	—	—	—	—	—	—
	ADDX，SUBX	B	B	—	—	—	—	—	—	—	—	—	—	—
	ADDS，SUBS	—	L	—	—	—	—	—	—	—	—	—	—	—
	INC，DEC	—	BWL	—	—	—	—	—	—	—	—	—	—	—
	DAA，DAS	—	B	—	—	—	—	—	—	—	—	—	—	—
	MULXU，MULXS，DIVXU，DIVXS	—	BW	—	—	—	—	—	—	—	—	—	—	—
	NEG	—	BWL	—	—	—	—	—	—	—	—	—	—	—
	EXTU，EXTS	—	WL	—	—	—	—	—	—	—	—	—	—	—
論理演算命令	AND，OR，XOR	BWL	BWL	—	—	—	—	—	—	—	—	—	—	—
	NOT	—	BWL	—	—	—	—	—	—	—	—	—	—	—
	シフト命令	—	BWL	—	—	—	—	—	—	—	—	—	—	—
	ビット操作命令	—	B	B	—	—	—	B	—	—	—	—	—	—
分岐命令	Bcc，BSR	—	—	—	—	—	—	—	—	—	○	○	—	—
	JMP，JSR	—	—	○	—	—	—	—	—	○	—	—	○	—
	RTS	—	—	—	—	—	—	—	—	—	—	—	—	○
システム制御命令	TRAPA	—	—	—	—	—	—	—	—	—	—	—	—	○
	RTE	—	—	—	—	—	—	—	—	—	—	—	—	○
	SLEEP	—	—	—	—	—	—	—	—	—	—	—	—	○
	LDC	BBW	W	W	W	—	W	W	—	—	—	—		
	STC	—	BW	W	W	W	—	W	W	—	—	—	—	
	ANDC，ORC，XORCB	—	—	—	—	—	—	—	—	—	—	—	—	
	NOP	—	—	—	—	—	—	—	—	—	—	—	—	○
	ブロック転送命令	—	—	—	—	—	—	—	—	—	—	—	—	BW

H8/3048Fデータシートp2-11「表2.2　命令とアドレッシングモードの組み合わせ」とp2-12 <<オペレーションの記号>>を引用

【記号説明】B：バイト，W：ワード，L：ロングワード

【注】*本LSIでは使用できません。

表1-1-b　命令の機能別一覧

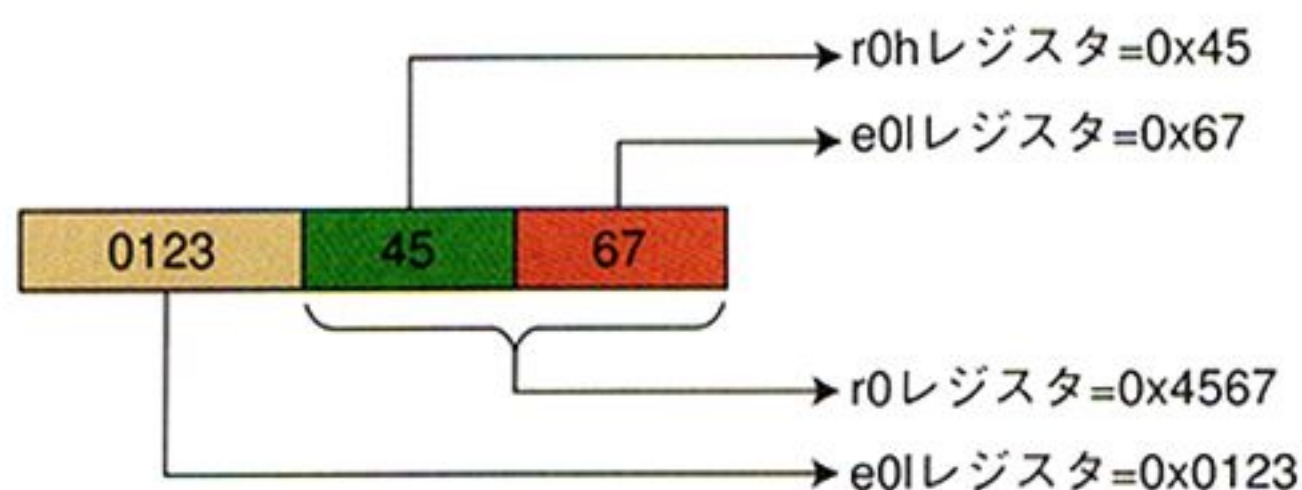
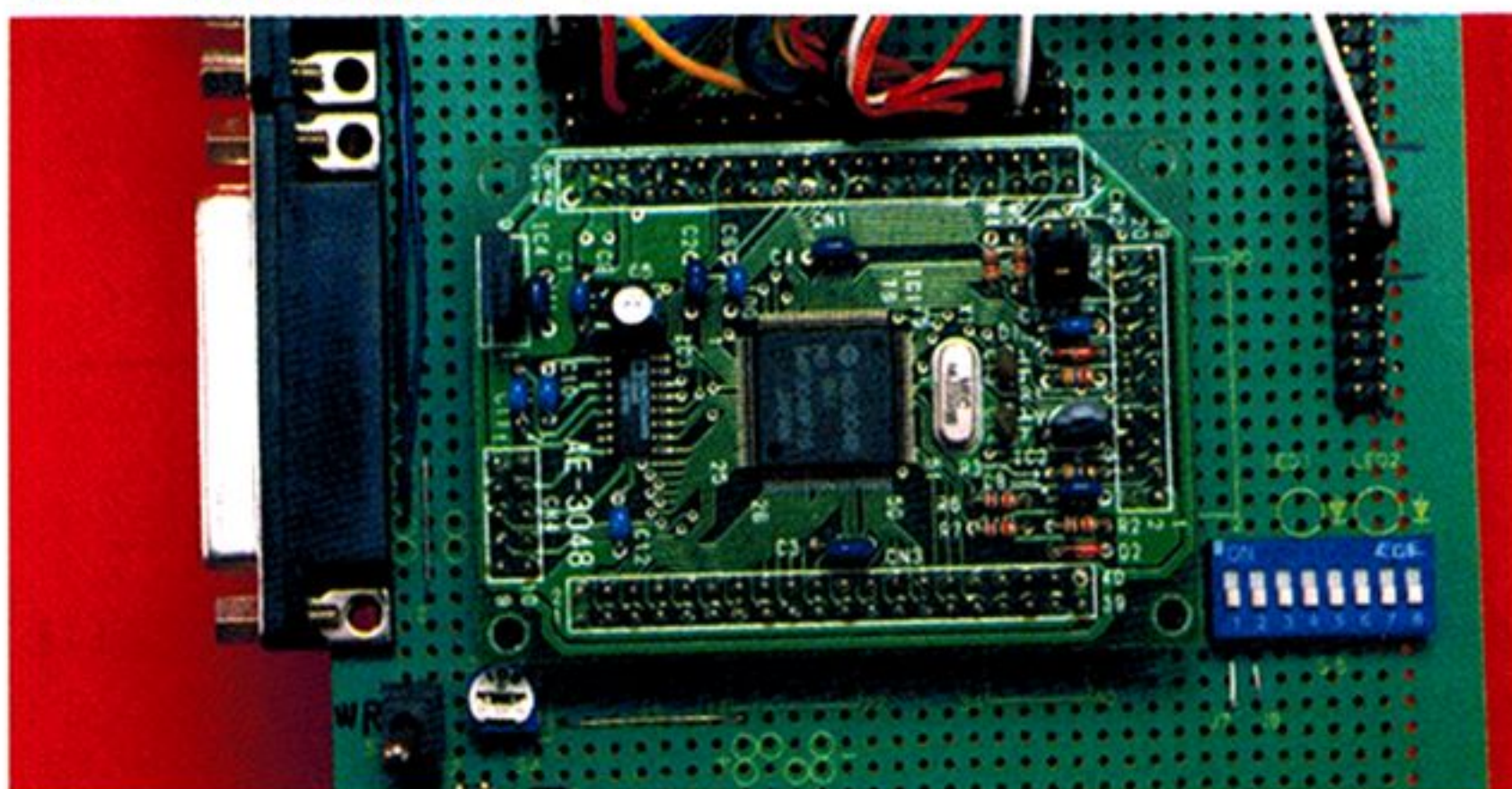
《オペレーションの記号》

Rd	汎用レジスタ（デスティネーション側）*
Rs	汎用レジスタ（ソース側）*
Rn	汎用レジスタ*
ERn	汎用レジスタ（32ビットレジスタ／アドレスレジスタ）
（EAd）	デスティネーションオペランド
（EAs）	ソースオペランド
CCR	コンディションコードレジスタ
N	CCRのN（ネガティブ）フラグ
Z	CCRのZ（ゼロ）フラグ
V	CCRのV（オーバフロー）フラグ
C	CCRのC（キャリ）フラグ
PC	プログラムカウンタ
SP	スタックポインタ
#IMM	イミディエイトデータ
disp	ディスプレースメント
＋	加算
－	減算
×	乗算
÷	除算
∧	論理積
∨	論理和
⊕	排他的論理和
→	転送
～	反転論理（論理的補数）
:3/:8/:16/:24	3/8/16/24ビット長

図1-1　H8のレジスタの使い方

写真1　中央部がAKI-H8ボード

図1-2a　H8/3048Fの動作モード

【注】* 内蔵RAMをディセーブルにすると外部アドレス空間になります

●レジスタ

32ビット長のレジスタが8本ある（er0 ～ er7。ただし，最後のer7はスタックポインタとして使用されるので，実質7本）。

これらのレジスタは上位ワード（e0～e7），下位ワード（ r0～r7）と分けて，16ビットのレジスタとしても使用可能 。

レジスタの下位ワードは，さらに上位バイト（r0h～r7h），下位バイト（r0l～r7l）と分けても使用可能。

これらをまとめると，図1-1のようになります。

16ビットアクセスを基本と考えれば，4本をアドレスレジスタとして使用したとしても，残りの4本のレジスタは上位ワード，下位ワードと分けて使えます。表1-1のニーモニック表からもわかるとおり，下位ワードレジスタでできることは，たいてい上位ワードレジスタでもできるようになっていますから，実質，4本のアドレスレジスタ＋8本の汎用レジスタが使えることになるわけで，MC68000（8本のアドレスレジスタ＋8本の汎用レジスタ）に近い感覚でプログラミングを行えます。

●アセンブラ疑似命令

プロセッサ自身の特徴ではないですが，アセンブラでの値の扱いが若干異なります。MC68000と同様に，即値は#で指定します。ただし，間接アドレッシングは，先頭に@を指定します。相対分岐命令（BRA：無条件，Bcc：条件付き，BSR:サブルーチンコール）は，#をつけなくてもアセンブラが勝手に即値としてラベルを解釈してくれますが，絶対分岐（JMP：無条件，JSR：サブルーチンコール）はなぜか間接アドレッシングとして扱わなければならない，という点もMC68000と同じです。

図1-2b　H8/3048Fの動作モード

モード5
(内蔵ROM 有効拡張1M バイトモード)
H'00000
ベクタエリア
H'000FF
内蔵ROM
H'07FFF
H'1FFFF
H'20000
H'3FFFF
H'40000
H'5FFFF
H'60000
H'7FFFF
H'80000
H'9FFFF
H'A0000
H'BFFFF
H'C0000
H'DFFFF
H'E0000
外部アドレス空間
エリア0
エリア1
エリア2
エリア3
エリア4
エリア5
エリア6
エリア7
H'F8000
H'FEF0F
H'FEF10
内蔵RAM*
H'FFF00
H'FFF0F
H'FFF10
外部アドレス空間
H'FFF1B
H'FFF1C
内部I/O レジスタ
H'FFFFF
メモリ間接分岐アドレス
絶対アドレス16ビット
絶対アドレス8ビット
絶対アドレス16ビット

モード6
(内蔵ROM 有効拡張16M バイトモード)
H'000000
ベクタエリア
H'0000FF
内蔵ROM
H'007FFF
H'FFEF0F
H'FFEF10
H'01FFFF
H'020000
H'3FFFFF
H'400000
H'5FFFFF
H'600000
H'7FFFFF
H'800000
H'9FFFFF
H'A00000
H'BFFFFF
H'C00000
H'DFFFFF
H'E00000
外部アドレス空間
エリア0
エリア1
エリア2
エリア3
エリア4
エリア5
エリア6
エリア7
H'F8000
H'FEF0F
H'FEF10
内蔵RAM*
H'FFF00
H'FFF0F
H'FFF10
外部アドレス空間
H'FFF1B
H'FFF1C
内部I/O レジスタ
H'FFFFF
メモリ間接分岐アドレス
絶対アドレス16ビット
絶対アドレス8ビット
絶対アドレス16ビット

モード7
(シングルチップアドバンストモード)
H'00000
ベクタエリア
H'000FF
内蔵ROM
H'07FFF
H'1FFFF
H'F8000
H'FEF10
内蔵RAM*
H'FFF00
H'FFF0F
H'FFF1C
内部I/O レジスタ
H'FFFFF
メモリ間接分岐アドレス
絶対アドレス8ビット
絶対アドレス16ビット

【注】*内 蔵RAM をディセーブルにすると外部アドレス空間になります。

例)

```
BRA    NearLabe
JSR    @FarLabe
```

●動作モード

H8シリーズのなかでも，H8/300Hと呼ばれるファミリ（今回のH8/3048Fも相当）は図1-2のように動作モードを切り替えられます。これは動作中にダイナミックに切り替えるよりは，MD0～2ピンをプルアップ/プルダウンしてアプリケーションごとに決める，というのが正しい使い方のようです。ほかの組み込みマイコンでも動作モードにより，内蔵メモリが見えたり見えなかったりするというのは多々あります。しかし，H8/300Hシリーズの場合はCPUのアドレス幅まで変えられてしまうというのがユニークです。x86のプロテクトモード，リアルモードの考えに近いともいえるかもしえません。

モニタプログラムの移植

AKI-H8の発売元である秋月電子通商では，ボードのほかにも「H8モニタデバッガ」という名前でプログラムを販売しています（2000円）が，最新のものでなかったり（Version 2.0c），モード7専用（アドレスバス/データバスを外部に出せない）だったりと，必ずしも今回の用途に最適なものではありません。後述のように，将来の拡張性を考えるとI/Oポートがたくさんあるよりは，外部メモリ空間にペリフェラルを拡張できたほうがシステムに柔軟性を持たせられますので，今回はH8/3048Fをモード6で使用します。

そこで各自でモニタプログラムを再ビルドしなければなりません。日立製

作所純正の開発ツール（アセンブラ＋リンカ＋ライブラリアン）を持っている場合はもう少し簡単に再ビルドできるのですが，ここではAKI-H8に付属の秋月電子製開発ツール（アセンブラ＋リンカ）しか持っていない場合の手順を紹介します。

再ビルドは以下のように行います。

1) 日立製作所のホームページ（http://www.hitachi.co.jp/Sicd/Japanese/Seminar/down.htm）に行き「サンプルソフト規約」をよく読んでください。
2) 「サンプルソフト規約」に同意すると「H8/300H　ダウンロードファイル」ページに着くので，「H8/300H用　モニタプログラム」をダウンロードします。
3) 展開したファイル中，advcmd/cmd24.srcというファイル中，100行目近辺を以下のように改造します。

改造前）

```
NOT.B   R6H     ; Test Check Sum
BNE     ERR11   ; Check Sum Error
```

改造後）

```
NOT.B   R6H     ; Test Check Sum
NOP             ; Check Sum Error
NOP
```

このファイルはPCからのHEXファイルをH8/3048Fが受け取るロードコマンド処理部の一部なのですが，この変更によりダウンロード時のチェックサム判定を無効にしています。どうも，日立製作所が考えるチェックサムの定義と秋月電子のとは食い違っているようです。

4) 本誌付録CD-ROMの以下のファイルを同じディレクトリにコピーしてください。
    pda/monitor/monitor.src（上書き）
    pda/monitor/build.bat
    pda/monitor/build.sub

これで，build.batをDOS Promptから実行すれば，数分後に，モニタプログラムを再ビルトできるようになります。

5) ビルドが成功したら，monitor.motというファイルが生成されているはずですので，AKI-H8に付属のFLASH.EXEを使ってこのファイルをデバイスに焼き込みます。

LCD編

LCDの仕組み

LCDを実際につなげる前にLCDの仕組みを調べてみましょう。3号の石上氏のタイマ製作のように，小規模なキャラクタタイプのLCDモジュールならば原理を気にする必要はないのですが，ある程度のグラフィックタイプになるとそうもいかないようです。大きめの書店へ行けば，LCDに関する専門書が何冊かあるのですが，大部分が液晶の物理的特性（異方性結晶分子における複屈折性うんちゃら，かんちゃら）から始まってしまうので，プログラマから見た視点で以下にまとめておきます。

1)　LCDの画素はマトリクス上に配置されている

LCDの各画素は，2つの金属板に挟まれて，金属板間に生じる電界により白/黒の表示を行います（実際はLCDパネル自身は光の偏光率を変えるだけで，その光が偏光板を通れるか否かが白/黒になるのですが，ここでは細かいことは気にしません）。各画素から信号線を引いてくることは現実的でないので，たいていはこの金属板は図2-1のように，X座標（セグメント電極），Y座標（コモン電極）で管理されます。

つまり（x, y）で画素を指定し，そこへ電圧をかけるか否かで白/黒の表示を行います。[*2]

2) 各画素は放っておくと消えてしまう

一定期間ごとにリフレッシュが必要です。普通はCRTと同様に，

図2-1　LCDパネルの構成

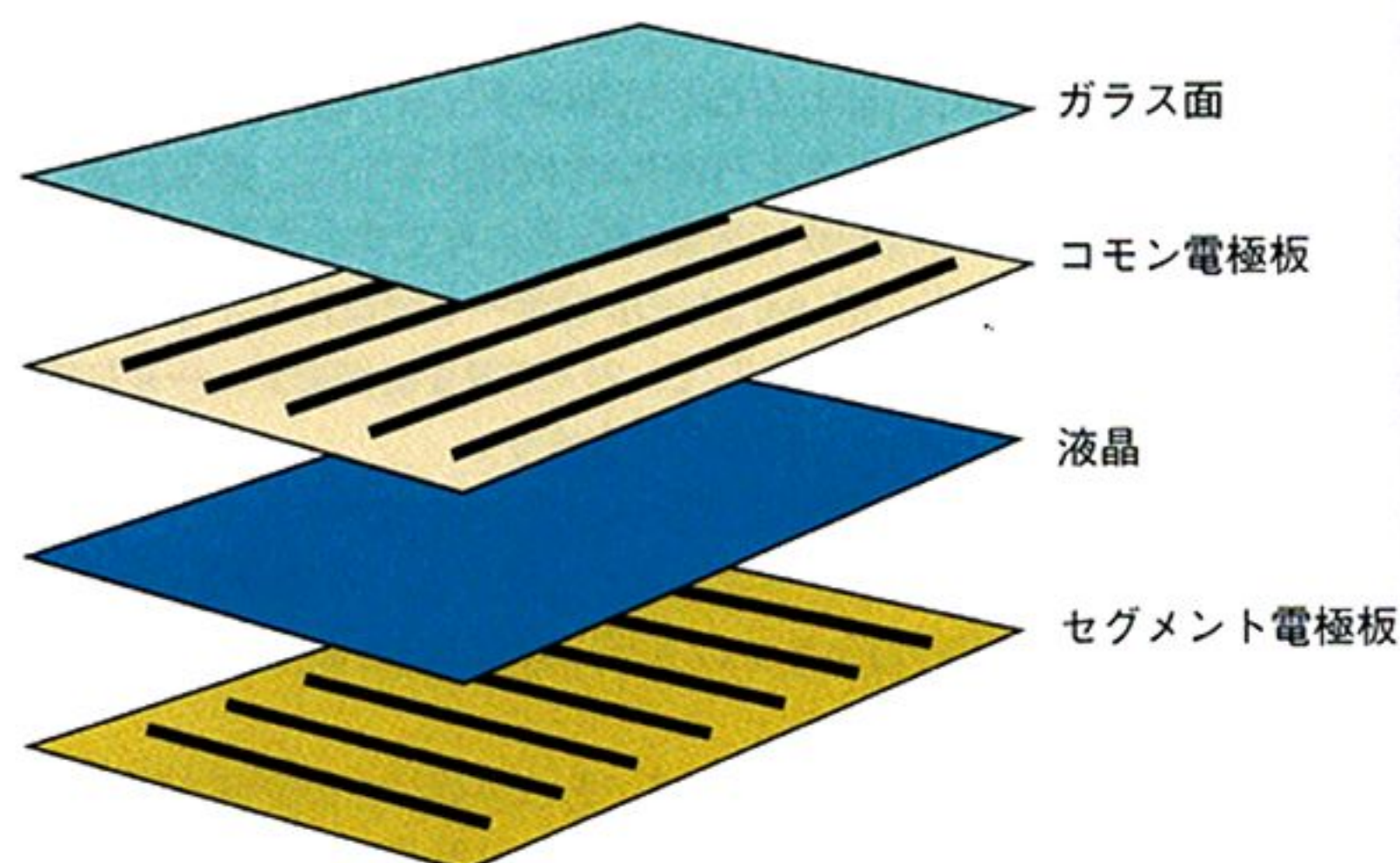



表2-1　CRTとLCDモジュールとの関連

CRT	LCDモジュール
ドットクロック	セグメントドライバ内のシフトレジスタへのクロック
H.Sync	・セグメントドライバ内のシフトレジスタへのリセット信号 ・コモンドライバへのインクリメント信号
V.Sync	コモンドライバへのリセット信号

2a) LCDパネルとは別に，画像情報を蓄えるメモリが必要（以下，このメモリをVRAMと呼びます）
2b) 一定期間ごとに，VRAMの情報を（内容に変更がなくても）転送しなければならない（以下，便宜的にリフレッシュと呼びます）
という仕組みが必要です。走査線が1本しかないCRTとは違い，1）で破綻しない限り，一度に複数の画素にリフレッシュを行えます。

通常はY座標（コモン電極）を1本指定し，横方向を一度にリフレッシュするようです。横方向といっても，64ドットや128ドットあるわけで，たいていのセグメントドライバはシフトレジスタを横ドット数分持っていて，VRAMからは1，4あるいは8ビットごとにセグメントドライバへ転送するようです。

ここで，LCDモジュールをLCDパネル＋セグメントドライバ＋コモンドライバとして見た場合，CRTとは表5-2のように対応づけられます（乱暴ないい方だなぁ）。

実際には，以下のような液晶特有の問題があり，それぞれに対応する仕組みが必要となってきます。

3) LCDパネルの駆動電圧は，複数必要

液晶の駆動電圧は，10～30〔V〕の電圧が必要（ドット数が増えるほど高くなるらしい）です。また，液晶素子に直流を与え続けると，簡単に焼き付いてしまうそうなので，駆動電圧は交流を用います。そこで液晶パネルを駆動するためには10～30〔V〕くらいの電圧が複数必要となってきます。

4) 電源監視機能

3）とも関連するのですが，電源投入直後，まだLCD部分全体が初期化されないうちにセグメントドライバから直流がLCDパネルに流れ込むと，LCDパネルが破損してしまいます。よって，電源が立ち上がってもLCDモジュール全体の初期化が終わるまで，セグメントドライバコモンドライバからLCDパネルに信号が発生しないように監視します。

*1：世の中には，携帯電話のLCDパネルのように四角い画素だけでなく，たとえば，電池のマークやアンテナのマークの画素もあったりしますが，点滅の原理は同じです。ただし，(x,y) 座標系だけでとらえられない操作法になるので，ここではとりあえず無視します。

LCDモジュール

以上のような原理でLCDは表示されるのですが，これらの要素を1から組み立てるのは大変です。たとえば128×64ドット表示のLCDパネルは，少なくともセグメントドライバとの接続が128箇所，コモンドライバとの接続が

64箇所あるわけです。たいていは，これらの関連する部分をまとめたモジュールが売っているので，これを利用させてもらいましょう。

ざっと見渡したところ，ほとんどのLCDモジュールは，LCDパネル，セグメントドライバ，コモンドライバの3つを内蔵していて，残りの要素を内蔵しているか否かで，何種類かに分類されるようです。また，3）は単なるDC-DCコンバータですし，4）もCPUのI/Oポートでなんとかなるでしょうから，LCDモジュールの選び方として，2）のVRAMがモジュールに内蔵されているか否かがひとつのキーになってきます。

ここで，LCDの原理とは直接関係ないのですが，LCDモジュールの分け方として，キャラクタ表示かグラフィック表示かという分け方があります。

これもひと昔前のPCのたとえでいうと，VRAMかG-RAMかということですね。

キャラクタ表示（VRAM）というのは，ある座標にどの文字を表示させるか，という方法でアクセスします。たとえば，1行目1文字目に「A」を表示という感じです。LCDモジュールが20×2文字をサポートしていたら，RAMは24バイトあれば足りるようになっています。実際には，CG（Character Generator）ROMと呼ばれるメモリが別にあり，「A」のドット情報が展開されて，LCDパネルに送り込まれます。この方法は文字を表示させる分には簡単で便利なのですが，

決められた文字以外扱うことができない（アルファベット＋カタカナしかサポートしていない，たいていのLCDモジュールでは漢字を表示することはできない）

決められた領域以外に文字を置けない。たとえば，サポートされている文字がすべて8×8ドットならば，（8×n，8×n）（n：整数）の位置にしか文字を書くことができません。文字を2ドット左にずらすとか，3ドット上に上げるということはできません。

図形が描画できない

という欠点があります。逆に，グラフィックタイプのものは，すべての画素が自由に扱える代わりに，

1文字表示するためにも，8×8 ドット分の情報を転送しなければならない

すべての画素情報を保存するための大きなメモリが必要

という欠点があります。

自由度と複雑さのトレードオフなのですが，やはり携帯ゲーム機を作るからにはグラフィックタイプでいきましょう。

原理的にLCDパネルのドット分だけVRAMの容量が必要ですので，大きな分解能を持ったLCDモジュールほど大きなメモリが必要ということになります。

松コース） VRAM内蔵タイプのLCDモジュール

LCD表示に必要なコンポーネントがすべてモジュールに含まれるということは，LCD初心者の私でも取り扱いが簡単なはずなので，まず，こちらを調べました。後述の理由からか，私の探した範囲では128×128ドットのものが最大のようです。これ以上のものはVRAM外付けタイプのものしかないようです。

竹コース） VRAM外付けタイプのLCDモジュール

メモリといえば，システム内でCPUが必ず持っています。これをLCDモジュールと共有できるか検討します。たとえば，240×160のパネルがあったとします。これをCPUのメモリからDMA転送してVRAMの代わりをするとします（往年のPC-8001がやっていた方法ですね）。毎秒60回書き換えるとして，

240×160×60＝2304000〔bit/sec〕

を送るわけです。現在，AKI-H8上のCPUは16〔MHz〕で動かしていますから，2〔Mbps〕のデータを送っているDMAとアクセス競合が起きると，著しくパフォーマンスが低下するおそれがあります（DMAとCPUでアクセス競合が起きると，DMAアクセスが優先されて，CPUが待たされる）。また，

240×160＝38400〔bit〕＝4800〔bytes〕

となって，H8/3048Fの内蔵メモリ（4Kバイト）をすでに超えています。あまり現実的ではありません。

次に，LCDコントローラの管理下にVRAMを配置する方法を考えます。LCDモジュールにVRAM用のソケットがあるわけではありませんから，システム構成は，

図2-2　G1216の内部ブロック回路

となり，VRAMだけでなくLCDコントローラも外付けになります。

実際には，VRAM内蔵のLCDコントローラも存在するらしいのですが，あまり一般的でないようです。アクセス速度が数十〔nsec〕とはいえ，数KBのSRAMを内蔵することは，簡単ではないようですね（マイコンがメモリを内蔵するのと同様に，システム面積は少なくなるし，消費電力も下がるはずなのですが……）

グラフィック型LCDを扱うのは今回が初めてですし，CPUボード上に2つも大型ICが増えるのは小型化する際に障害になりかねないので，今回はVRAM内蔵型のLCDモジュールを採用しました。

セイコーインスツルメンツ社製G1216

大きさで選べば，128×128ドットのものもあるのですが，システムを組み上げたとき正方形では不格好のような気がしますし，それほどメリットも感じませんので，今回はセイコーインスツルメンツ社のG1216（128×64ドット）を使用します。

これはLCDパネルのほかに，

バイアス電位生成回路：LCDパネルは5〔V〕や3.3〔V〕くらいでは動きません。15〔V〕とか17〔V〕とか，いろいろな電圧で動かすみたいですが，それらを自動生成する回路が内蔵されています。ただし，負電圧を生成する機能はないので，-8〔V〕を外部から与える必要があります。

S-RAM付きセグメントドライバ：LCDパネルは網目状になっていて，縦横方向から制御し，電圧が生じたところが黒く写るのですが，セグメントドライバは横方向からの信号を制御します。

コモンドライバ：LCDパネルの縦方向の信号を制御します。

と，LCD制御に必要な回路がひととおり入っているという使いやすいモジュールです。

図2-3にG1216のブロック図を示します。図を見るとわかるとおり，セグメントドライバが2つ入っていて，それぞれからチップセレクト信号が出て

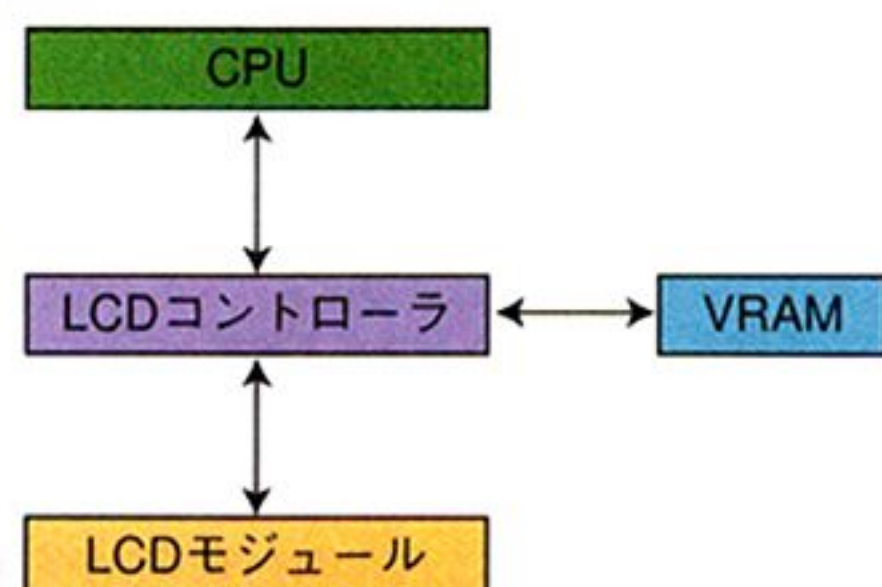



写真2　セイコーインスツルメンツG1216

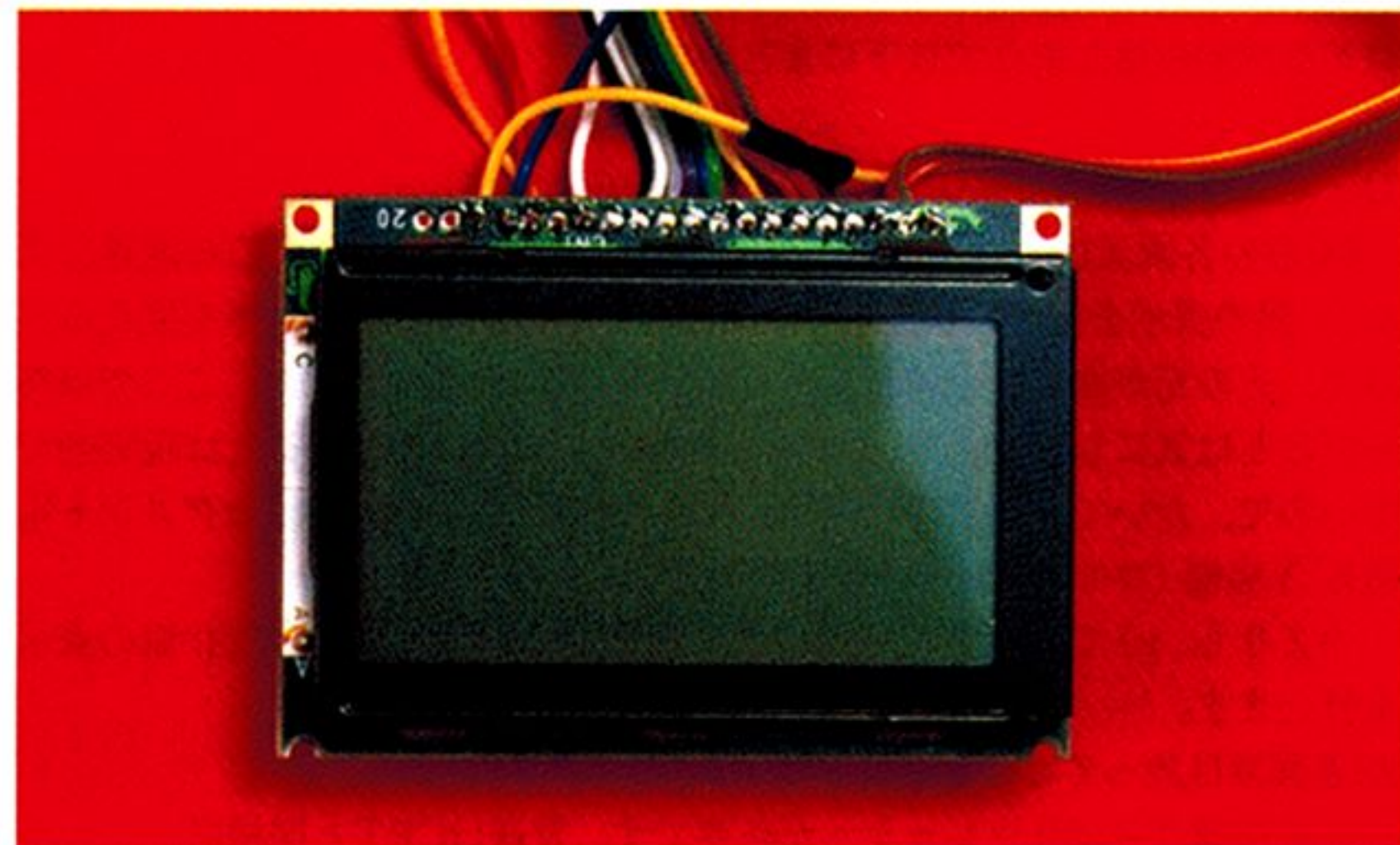



図2-3　G1216の内部ブロック回路

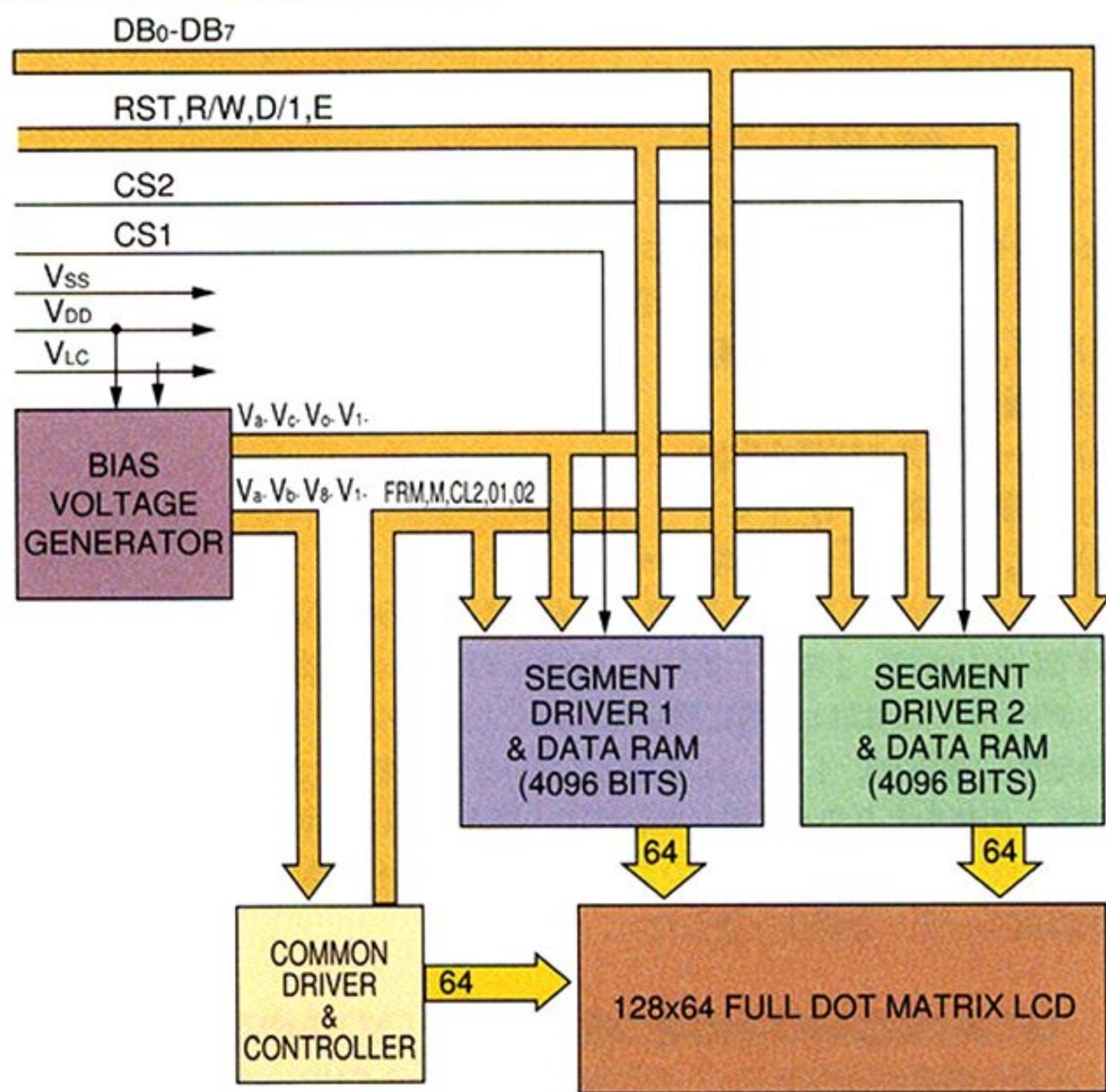
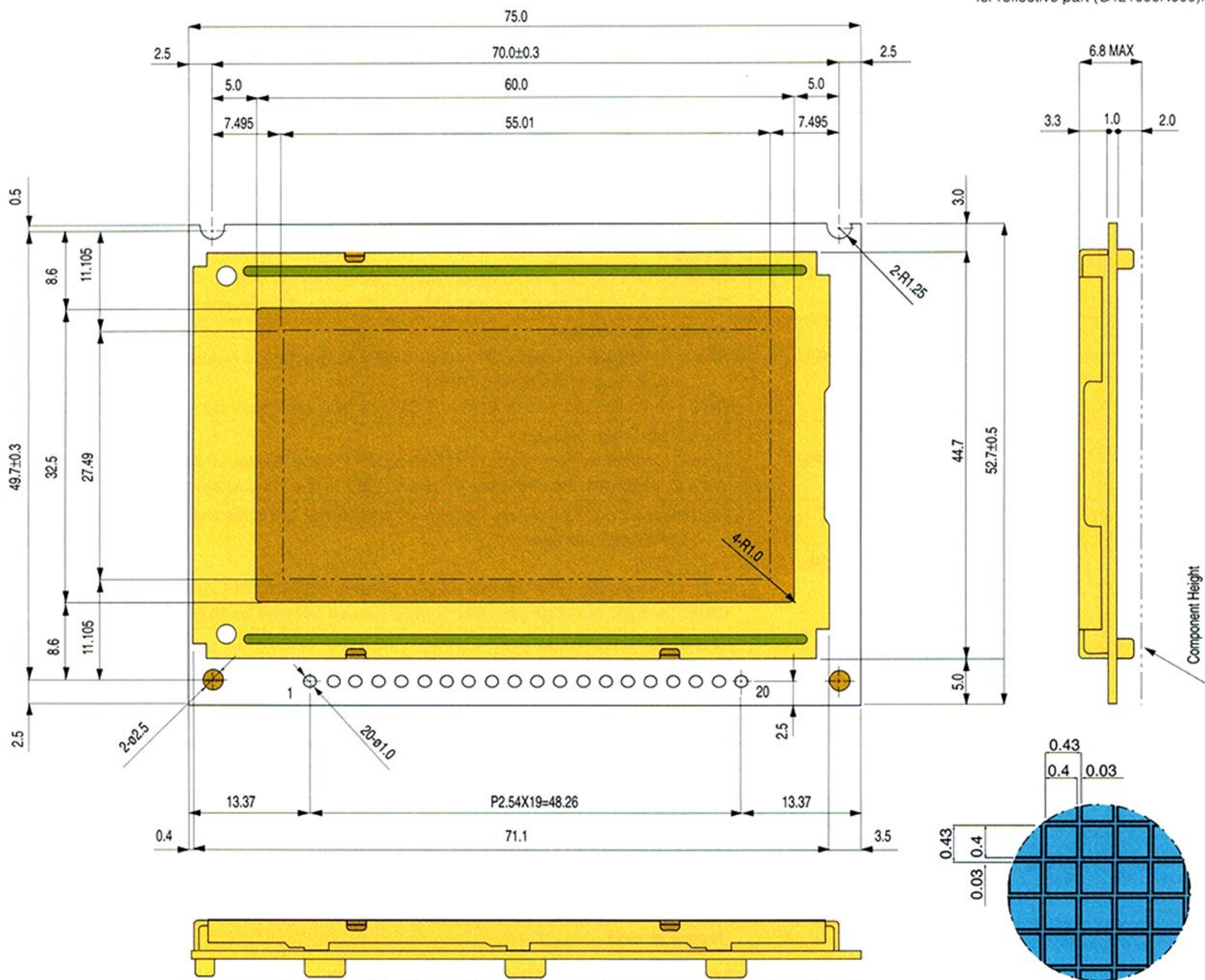



図2-4　G1216の外形寸歩図

CN1 G1216			
No.	Symbol	No.	Symbol
1	V_{DD}	11	DB7
2	V_{SS}	12	CS1
3	V_{LC}	13	CS2
4	DB0	14	RST
5	DB1	15	R/W
6	DB2	16	D/I
7	DB3	17	E
8	DB4	18	F_{GND}
9	DB5	*19	LED (+)
10	DB6	*20	LED (-)

*No connection to pins 19 & 20 for reflective part (G121600N000).

います。つまり，このLCDモジュールは128×64の表示を行いますが，内部的には，64×64のLCDモジュールが2つ横に並んで入っていることと同じです。実際，プログラム内部でもLCDパネルのどこに描画を行うかによってどちらのセグメントドライバへアクセスするかを切り分けています。

G1216とAKI-H8をつなぐ(その1：I/Oポート編)

LCDモジュールが決まったところで，さっそくAKI-H8につなげて遊んでみましょう，と思ったのですが，G1216のピン端子表を見て愕然としました。「E信号」が要求されているではないですか。この「E信号」とは，その昔，CPUがまだ1〔MHz〕や2〔MHz〕で動いていた時代，モトローラ系のCPU（6802や6809など）で使われていた「Eクロック」信号です。ストローブ信号（$\overline{AS}$など）の立ち下がりではアドレスバスが安定していないのでEクロックの立ち上がりまで待ってね，というにものでした。

私が知る限り，MC68000が最後の「Eクロック」を搭載したCPUでした。後継のMC68020/68030ではサポートされていない信号だったので，X68000用に68030アクセラレータを作成した際，この「Eクロック」をどう作成しようかと悩んだこともあるのですが，「Eクロック」を必要とするような古い周辺デバイスはX68000に使われていなかったということもありました。

もちろん，H8/3048にはそんな信号はありません。G1216のデータシートには図2-6のような参考例があって，最初は，なぜZ80-PIOが必要なのだろう？　Z80のバスに直結すればいいのに，と思っていましたが，きっとデータシートを書いた人が考え及ばなかった違いないと勝手に無視していました。自分が配線するときになって，Z80にはEクロックがないからバス接続にはできないののだということがわかりました。つまり，参考回路は，バスアクセス/制御信号などをすべてZ80-PIOを通して行うことにより，擬似的に「Eクロック」を生成していたのです。

仕方がないので，

Port 4:　データの入出力
Port B　制御信号

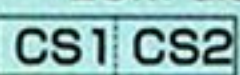
B0：$\overline{CS1}$
B1：$\overline{CS2}$
B2：R/$\overline{W}$
B3：D/$\overline{I}$
B4：E信号

と割り振って，擬似的にアクセスします。

たとえば，Eクロックを伴う書き込み（CPU→LCD）では，

1) $\overline{RD}$, $\overline{CS}$, D/$\overline{I}$ 信号を確定
2) E信号を立ち上げ
3) データバス上に値を出力
4) E信号の立ち下げ
5) $\overline{RD}$, $\overline{CS}$, D/$\overline{I}$ 信号のネゲート

という手順を踏まなければなりません。これらの信号は自動生成されるわけではないので，1ステップずつ，H8のI/O Portから出力します。

データバス上は1）以前に確定していてもよいので，レジスタの有効活用のため，実際は3，1，2，4，5と行っています。また，3)と4)の間でデータバス上の値をとりあえず読んでおけば，読み出しにも同じサブルーチンを使えるので，実際にはリスト1のように読み出しにも書き込みにも使えるE信号作成ルーチンを制作しました。

G1216とAKI-H8をつなぐ(－10〔V〕を作る)

必要な機能がほぼモジュールに統合されているG1216ですが，負電源だけは内部で生成しません。モジュールに内蔵するよりもシステムから適当に使い回してきてね，ということなのかもしれません。今回使用したAKI-H8には，負電源はありませんからLCDモジュール用に作成します（データシートによると，1.8〔mA〕@－8〔V〕必要とのこと）。

このような用途にはマキシム社よりMAX680という便利なICが発売されているので，これを使用します。このICは，

1) 2つのコンデンサに，システム電源からの+5〔V〕を充電します

表2-2　TERMINAL FUNDTIONS

Signal	Qty	I/O	Destination	Functions
DB_0 to DB_7	8	I/O	MPU	Common terminal for tristate input and output, and data bus. DB7 = MSB
E	1	Input	MPU	Enable Write (R/W = 0): Latches data of DB_0 to DB_7 at the fall of E. Read (R/W = 1): Outputs data to DBo to DB_7 while "E" keeps a high level.
R/W	1	Input	MPU	Read/Write selection R/W = 1: When E = 1 and CS1 = 0 or CS2 = 0, the data is output to DB_0 to DB_7 and read is available by MPU. R/W = 0: When CS1 = 0 or CS2 = 0, DBo to DB7 are ready for receiving the input.
D/I	1	Input	MPU	Data Instruction selection D/I = 1: Indicates that the data in DBo to DB7 is the display data. D/I = 0: Indicates that the data in DBo to DB7 is the instruction code.
CS1, CS2	2	Input	MPU	Chip select input Active low. Data input and output is possible under the following status: LCM display screen CS1 CS2 CS1: Controls the LCM left half display screen (SEG1 to SEG64). CS2: Controls the LCM right half display screen (SEG65 to SEG128).
RST	1	Input	MPU	Reset signal (Active low). Setting the RST signal to a low level allows for initial setup. (1) ON/OFF register: 0 setup (display OFF) (2) Display start line register: 0 line setup (display starts from 0 line) The setup status is retained until the status is changed by an instruction after reset is released.
V_{DD}	1	-	Power	Power terminal for logic (+5V)
V_{SS}	1	-	Power	GND terminal (0V)
V_{LC}	1	-	Power	Power terminal for LC drive
LEDA	1	-	Power	LED backlight anode terminal (+)
LEDC	1	-	Power	LED backlight cathode terminal (-)
F_{GND}	1	-	-	Frame ground[1]

F_{GND} terminal is connected to the metallic frame of the module. Use this terminal when grounding the frame.

図2-5 G1216のアクセスタイミング表

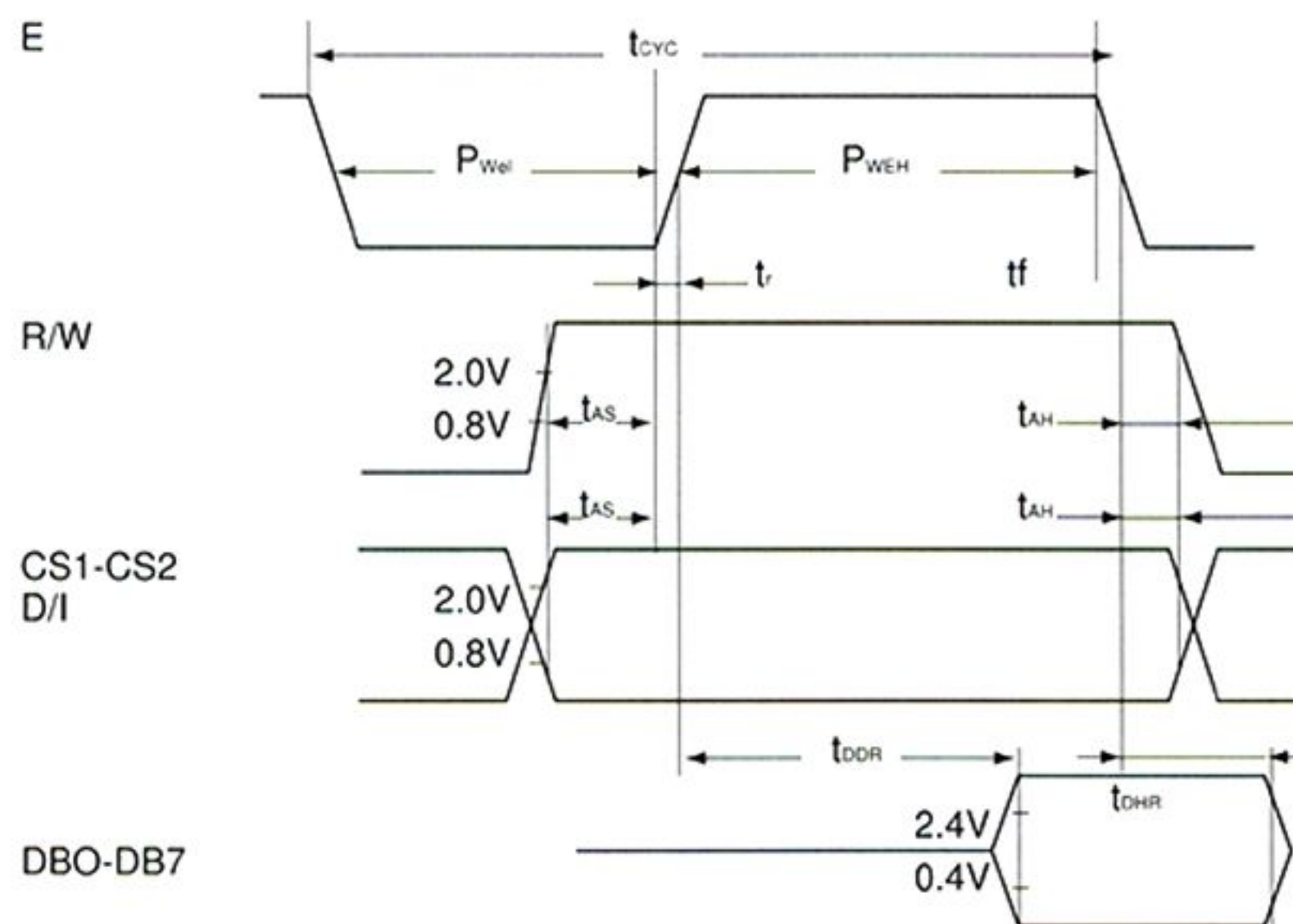
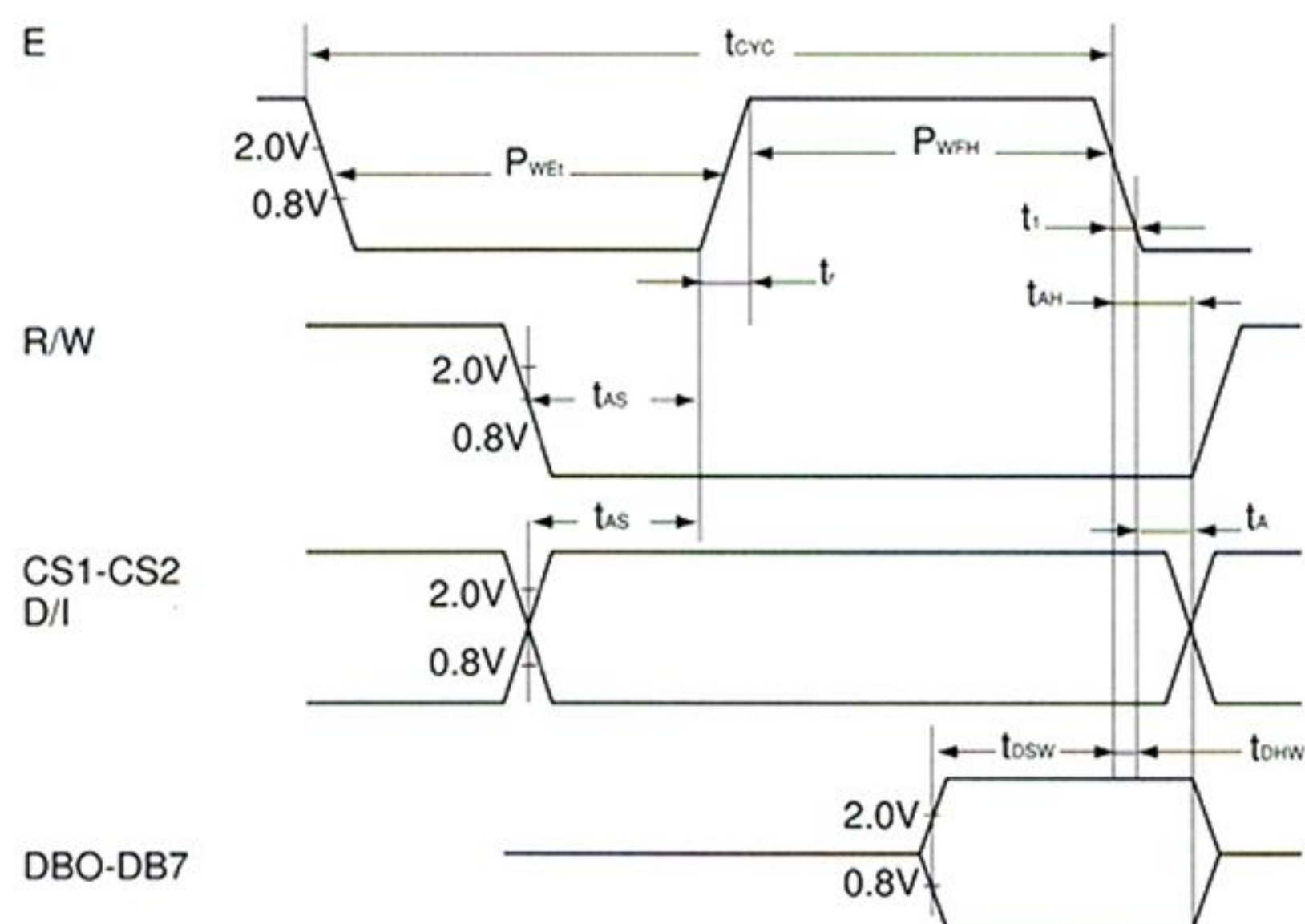
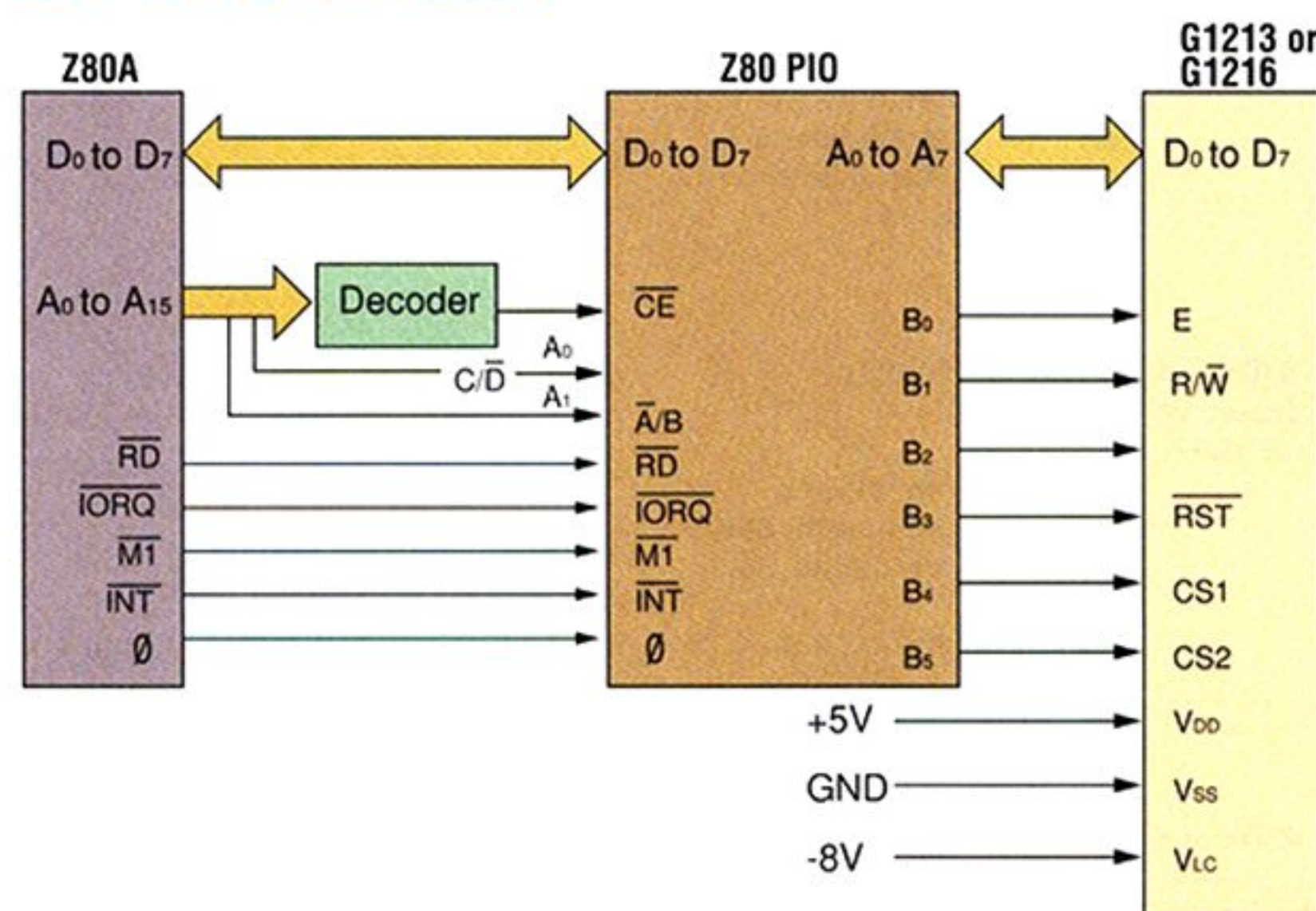



図2-6 Z-80とG1216の配線例

2）充電完了後，システム電源からコンデンサを切り離します
3）充電されたコンデンサの-側を，システムの+5〔V〕へ接続します。すると，コンデンサの＋側は，5＋5＝＋10〔V〕で，システムの0〔V〕に対して，＋10〔V〕を出力していることになります

この1～3を高速に繰り返すことにより，電源電圧の倍の電圧を生成します。

1）で2つのコンデンサを同時に充電し，3）で逆向きに放電させれば，-10〔V〕も容易に作成できます。

なお，データシートによると姉妹品のMAX681はチャージポンプ用のコンデンサも内蔵しているそうですので，もし入手できたら，そちらのほうがもっと簡単に小さくシステムを組み上げることが可能です。

こうして作った-10〔V〕を，このまま，G1216のVLCピンへ接続してもよいのですが，コントラスト調整用に20〔KΩ〕の半固定抵抗を入れましょう。

また，現在はDCアダプタからの+5〔V〕を電源として用いていますが，3.3〔V〕で乾電池駆動するかもしれません。そのようなときでも，MAX680は入力電圧として2～6〔V〕をサポートしていますし，カスケード接続もOKなので，3.0〔V〕×（+2）×（−2）＝−12〔V〕で，−8〔V〕を得るためには問題ないでしょう（その前に3.3〔V〕で動くLCDモジュールを探したほうがよいかもしれませんね）。

G1216とAKI-H8をつなぐ（その2：メモリマップド編）

今回の企画は携帯ゲーム機を作ってみる，ということだったので（ゲームプログラムそのものがないのはともかく），CPU＋キーボード＋LCDモジュールをもって完成としてもよいのですが，やはり，ゲームだけでなく簡易PDAとしても使いたいものです。個人的に，PDAには和英・英和辞典は必須と思っていますし，漢字も（入力はともかく）出力くらいはできないとかっこがつきません。また，H8/3048F内蔵のフラッシュメモリは100回までの書き込みしか保証されていないので，これとは別の不揮発性メモリが外部にほしいところです。

そういうわけで，次回の予告を書きながら思ったのですが，このままでは，ピンが足りません。H8/3048は12ポートもIOピンを持っているのですが，すべてがなんらかのピンとマルチプレクスされています（表1-2）。たとえば，ポート9はSIOとマルチプレクスされていますが，SIOはデバッグに必須なのでI/Oとしてのポート9はあきらめなければなりません。

仮に1M Bitのフラッシュメモリを外付けにしようとすると，

ポート1	アドレスバス0～7として使用
ポート2	アドレスバス9～15として使用
ポート3	データバス8～15として使用
ポート5	アドレスバス16～20として使用
ポート6	制御信号（$\overline{RD}$, $\overline{HWR}$, $\overline{LWR}$, $\overline{AS}$など）として使用

と半分くらいのポートが使えないことがわかります。さらにキーボードスキャン用に3ポート（8×10のマトリクスなので，正確には，8＋10＝18ビット）とぎりぎりです。そのほかにもEEPROM（住所録/メモ帳などの記録用）もつけたいですし，せっかくですから着メロくらいは鳴らしたいですし，電池の残量も見れるといいなぁ，とその他にも拡張する用途はいくらでもあるのにピンが足りなくなってしまいます。

これは困りました。せめて，E信号さえH8/3048が自動生成できたら，LCDモジュールをメモリ空間へ移せるのにと思いながら，ふと，E信号を外してみると，半分くらい，まともに動きます。さらにプルアップしてみると，あれ？　まったく問題なく動作します。

G1216のデータシートを見ると，

CPUからの書き込み：$\overline{CS}$=0，R/$\overline{W}$=0のとき，E信号の立ち下がりで，データバス上の値がG1216に書き込まれる
CPUからの読み出し：$\overline{CS}$=0，R/$\overline{W}$=1のとき，E信号がHighの間，データバス上に，G1216からの値が出力される

とあります。実際，タイミングチャートを見ると，R/$\overline{W}$信号，$\overline{CS}$信号，D/$\overline{I}$信号のセットアップタイムは，E信号の立ち上がり，データバスの確定はE信号の立ち下がりを基準に規定されています。これらの信号を読み込むためにはE信号が変化しなければいけないはずなのですが，なぜかLCDモジュールは正しく動作します。

データシートをさらに詳しく読んでみると，各ピンの機能説明のところに，

E信号：
書き込み（R/$\overline{W}$＝0）：立ち下がりで，データバス上の値をラッチする
読み出し（R/$\overline{W}$＝1）：Highの間，データバス上に値を出力する

と書いてあります。書き込み時，E信号がHighの間は，どうなるかは触れられていませんが，おそらくデータバス上の値をLCDモジュールへ筒抜けさせているのでしょう。

となると，残された トリガー信号は，$\overline{CS}$かR/$\overline{W}$ということになります

リスト1　H8のIOポートを使用したときのG1216制御ルーチン　port.mar

```
LCD_Dir             .equ       h'ffffc5   ; Port 4 Direction 0:input,
1:output
LCD_Data   .equ     h'ffffc7   ; Port 4 Data
LCD_Cnt             .equ       H'ffffd6   ; Port B

;
; 左のモジュールがready状態になるまで待機
;
LBusy:
          mov.b      #h'00,r0h
          mov.b      r0h,@LCD_Dir           ; Input LCD Data

LBusy1:
          MOV.B    #H'86,R0L      ; E=0, D/_I=0, R/_W=1, _CS2=1, _CS1=0
          bsr        E_Sequence
          AND.B    #H'80,R0H
          BNE        LBusy1

          RTS

;
; 右のモジュールがready状態になるまで待機
;
RBusy:
          mov.b      #h'00,r0l
          mov.b      r0l,@LCD_Dir           ; Input LCD data

RBusy1:
          MOV.B    #H'85,R0L      ; E=0, D/_I=0, R/_W=1, _CS2=0, _CS1=1
          bsr        E_Sequence
          AND.B      #H'80,R0H
          BNE       RBusy1

          RTS
;
; 左のモジュールにコマンドを出力
; Input r0l: command
LCom_Out:
          mov.b      #h'ff,r0h
          mov.b      r0h,@LCD_Dir           ; Output LCD Data
          MOV.B      R0L,@LCD_Data

          MOV.B      #H'82,R0L  ; E=0, D/_I=0, R/_W=0, _CS2=1, _CS1=0
          bsr        E_Sequence
          rts

;
; 右のモジュールにコマンドを出力
;
; Input r0l: output data
RCom_Out:
          mov.b      #h'ff,r0h
          mov.b      r0h,@LCD_Dir           ; Output LCD Data
          MOV.B      R0L,@LCD_Data

          MOV.B      #H'81,R0L  ; E=0, D/_I=0, R/_W=0, _CS2=0, _CS1=1
          bsr        E_Sequence
          rts

;
; 左のモジュールにデータを出力
;
LDdat_Out:
          BSR        LBusy

          mov.b      #H'ff,r0l  ; Output LCD_Data
          mov.b      r0l,@LCD_Dir

          MOV.B      @e_dat,R0L
          MOV.B      R0L,@LCD_Data

          MOV.B      #H'8A,R0L  ; E=0, D/_I=1, R/_W=0, _CS2=1, _CS1=0
          bsr        E_Sequence
          rts

;
; 右のモジュールにデータを出力
;
RDat_Out
          BSR        RBusy

          mov.b      #H'ff,r0l  ; Output LCD_Data
          mov.b      r0l,@LCD_Dir

          MOV.B      @e_dat,r0h
          MOV.B      R0h,@LCD_Data

          MOV.B      #H'89,R0L  ; E=0, D/_I=1, R/_W=0, _CS2=0, _CS1=1
          bsr        E_Sequence
          rts
;
; Cordinate Memory access with "E signal"
;       in:       r0l       the start pattern
;       out:      r0h       read data (valid only for read access)
;
E_Sequence:
          MOV.L      #LCD_Cnt,ER1
          MOV.B      R0L,@ER1
          OR.B       #H'10,R0L  ; Enable E
          MOV.B      R0L,@ER1
          mov.b      @LCD_Data,r0h            ; Read data
          and.b      #H'0f,r0l  ; Disalbe E
          MOV.B      R0L,@ER1
          or.b       #H'0f,r0l  ; Set 1 for D/_I, R/_W, _CS0, _CS1
          MOV.B      R0L,@ER1
          RTS
```

図2-7　MAX680の簡単な原理

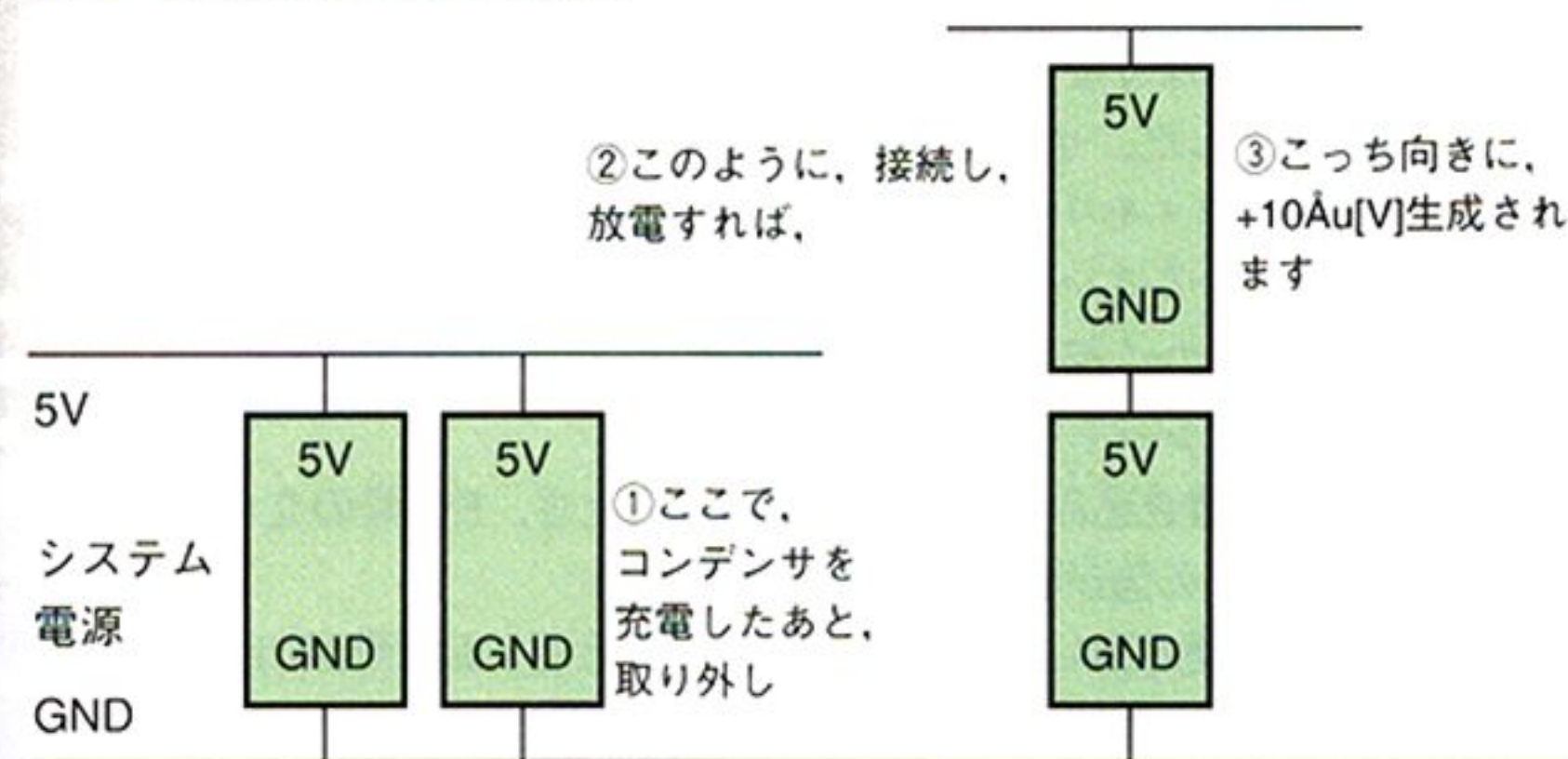
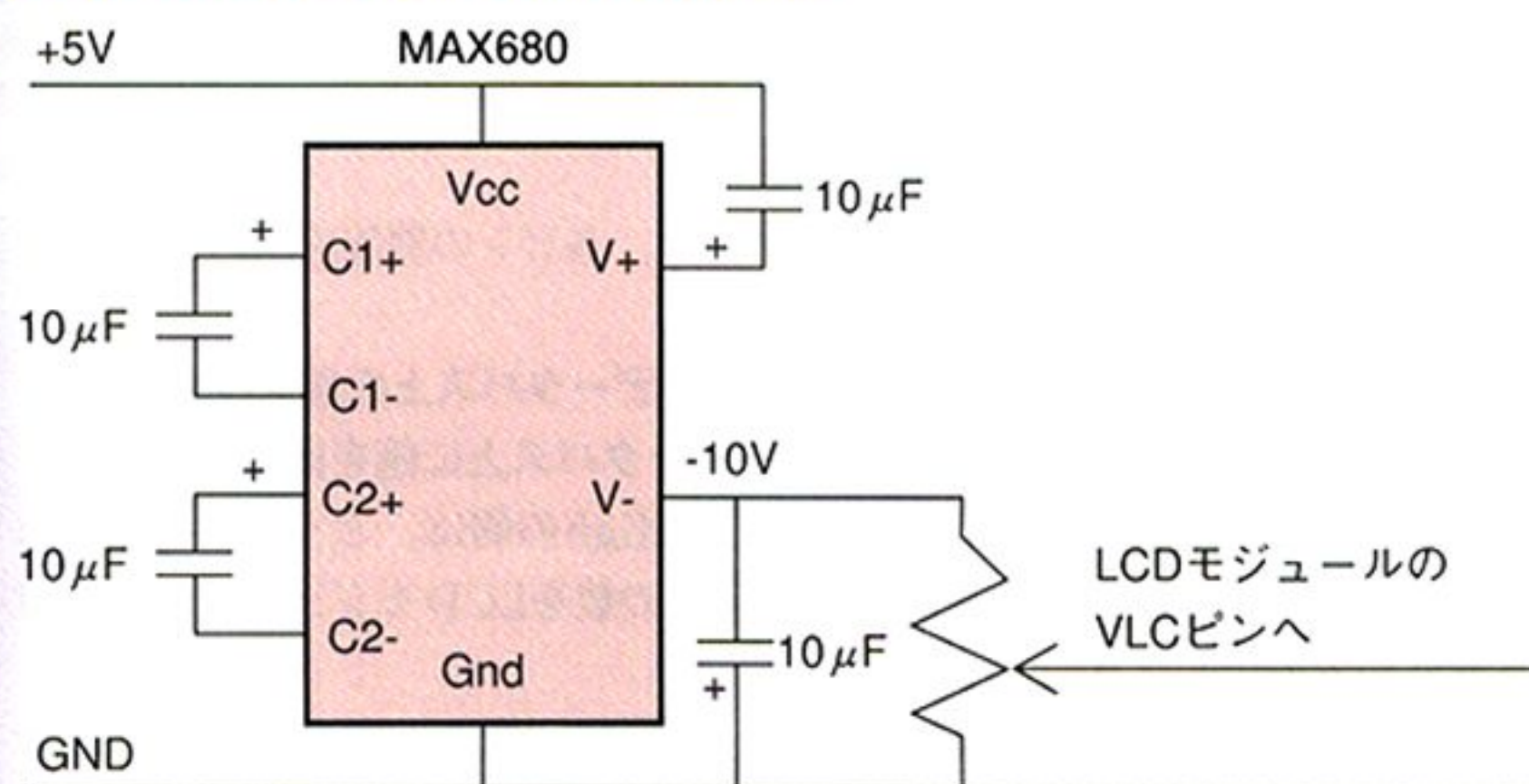
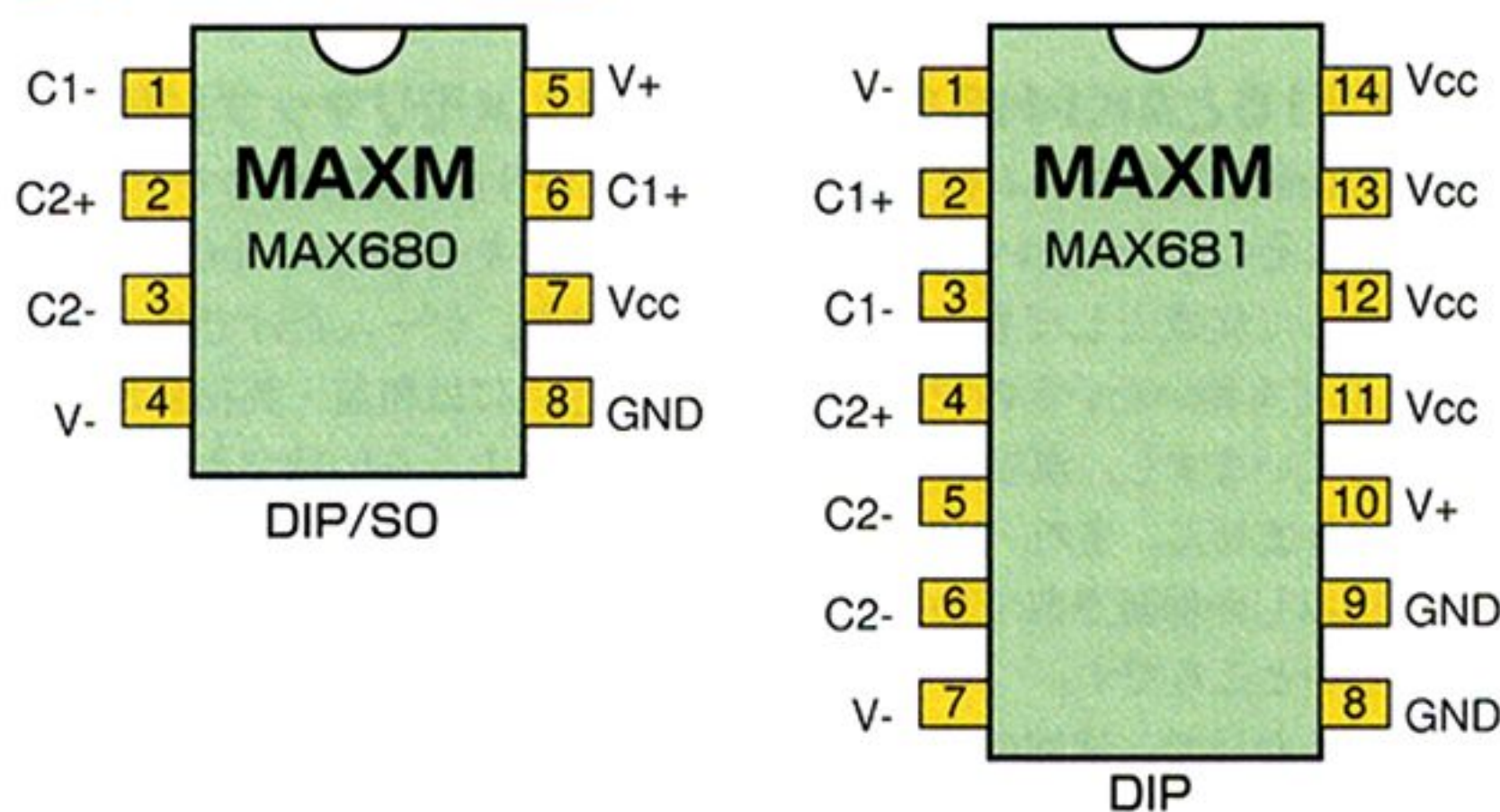



図2-9　MAX680のピン配置図

図2-8　G1216を使用するための昇圧回路

が，H8/3048はどちらに信号に対しても，データバスはホールド時間を持っています。

結論からいうと，G1216の本当のトリガー信号は_CSのようで，R/_W信号をH8の$\overline{\text{HWR}}$信号へつないでもうまく書き込みができませんでした。どうも，$\overline{\text{CS}}$の立ち下がり時にはR/$\overline{\text{W}}$信号が確定していないといけないようです。H8は，$\overline{\text{CS}}$信号に対して半クロック近く遅れてHWR信号が確定しますから，2つの信号をつないでも動作しないはずです。*2

このR/$\overline{\text{W}}$信号は，Port Bの5ビット目を接続し，別に制御させることにしました。

前述したように，G1216の$\overline{\text{CS}}$信号は左半分用と右半分用に2種類あるのですが，それぞれH8/3048の$\overline{\text{CS2}}$と$\overline{\text{CS3}}$を接続しました。H8/3048のモード

表1-2-a　動作モード別ポート機能一覧（1）

ポート	概要	端子	モード1	モード2	モード3	モード4	モード5	モード6	モード7
ポート1	・8ビットの入出力ポート ・LED駆動可能	$P1_7$～$P1_0$/A_7～A_0	アドレス出力端子（A7～A0）				アドレス出力端子（A7～A0）と入力ポートの兼用 DDR=0のとき入力ポート DDR=1のときアドレス出力端子		入出力ポート
ポート2	・8ビットの入出力ポート ・入力プルアップMOS内蔵 ・LED駆動可能	$P2_7$～$P2_0$/A_{15}～A_8	アドレス出力端子（A15～A8）				アドレス出力端子（A15～A8）と入力ポートの兼用 DDR=0のとき入力ポート DDR=1のときアドレス出力端子		入出力ポート
ポート3	・8ビットの入出力ポート	$P3_7$～$P3_0$/D_{15}～D_8	データ入出力端子（D15～D8）						入出力ポート
ポート4	・8ビットの入出力ポート ・入力プルアップMOS内蔵	$P4_7$～$P4_0$/D7～D_0	データ入出力端子（D7～D0）と8ビットの入出力ポートの兼用 8ビットバスモードのとき入出力ポート 16ビットバスモードのときデータ入出力端子						入出力ポート
ポート5	・4ビットの入出力ポート ・入力プルアップMOS内蔵 ・LED駆動可能	$P5_3$～$P5_0$/A_{19}～A_{16}	アドレス出力端子（A_{19}～A_{16}）				アドレス出力端子（A_{19}～A_{16}）と4ビットの入力ポートの兼用 DDR=0のとき入力ポート DDR=1のときアドレス出力端子		入出力ポート
ポート6	・7ビットの入出力ポート	$P6_6$/LWR $P6_5$/HWR $P6_4$/RD $P6_3$/AS	バス制御信号出力端子（LWR, HWR, RD, AS）						入出力ポート
		$P6_2$/BACK $P6_1$/BREQ $P6_0$/WAIT	バス制御信号入出力端子（BACK、BREQ、WAIT）と3ビットの入出力ポートの兼用						
ポート7	・8ビットの入出力ポート	$P7_7$/AN_7/DA_1 $P7_6$/AN_6/DA_0	A/D変換器のアナログ入力端子（AN7、AN6）およびD/A変換器のアナログ出力端子（DA1、DA0）と入力ポートの兼用						
		$P7_5$～$P7_0$/AN_5～AN_0	A/D変換器のアナログ入力端子（AN5～AN0）と入力ポートの兼用						
ポート8	・5ビットの入出力ポート	P84/CS0	DDR=0のとき入力ポート DDR=1のとき（リセット後）CS0出力端子						入出力ポート
	・$P8_2$～$P8_0$はシュミット入力	$P8_3$/CS_1/IRQ_3 $P8_2$/CS_2/IRQ_2 $P8_1$/CS_3/IRQ_1	IRQ3～IRQ1入力端子、CS1～CS3出力端子と入力ポートの兼用　IRQ_3～IRQ_0 DDR=0のとき（リセット後）入力ポート DDR=1のときCS1～CS3出力端子						入力端子と入出力ポートの兼用
		$P8_0$/RFSH/IRQ_0	IRQ_0入力端子、RFSH出力端子と入出力ポートの兼用						
ポート9	・6ビットの入出力ポート	$P9_5$/SCK_1/IRQ_5 $P9_4$/SCK_0/IRQ_4 $P9_3$/RxD_1 $P9_2$/RxD_0 $P9_1$/TxD_1 $P9_0$/TxD_0	シリアルコミュニケーションインタフェイスチャネル0、1（SCI0、1）の入出力端子（SCK_1、SCK_0、RxD_1、RxD_0、TxD_1、TxD_0）、およびIRQ_5、IRQ_4入力端子と6ビットの入出力ポートの兼用						

写真3　基板の配線例その1

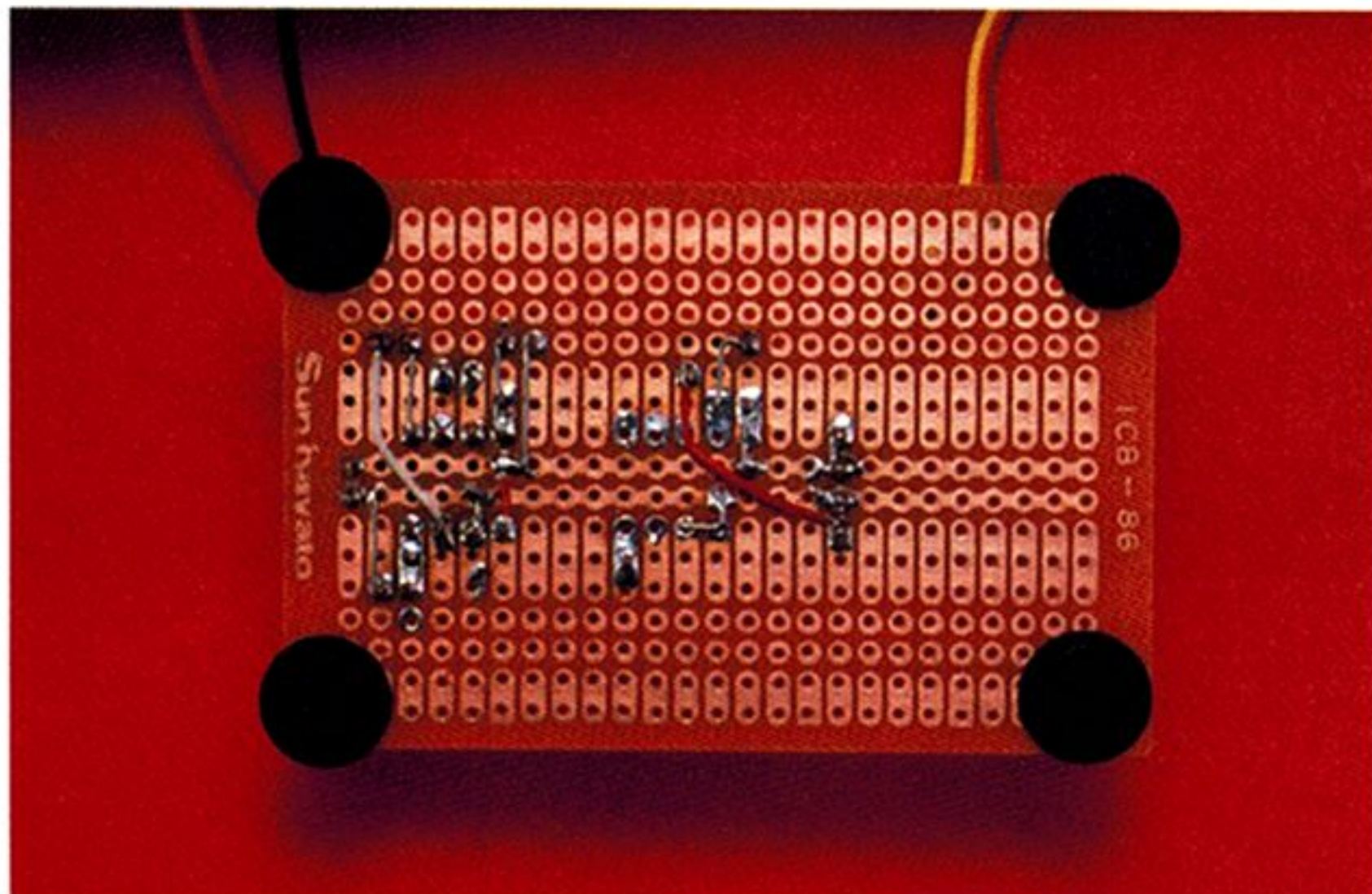



写真4　基板の配線例その2

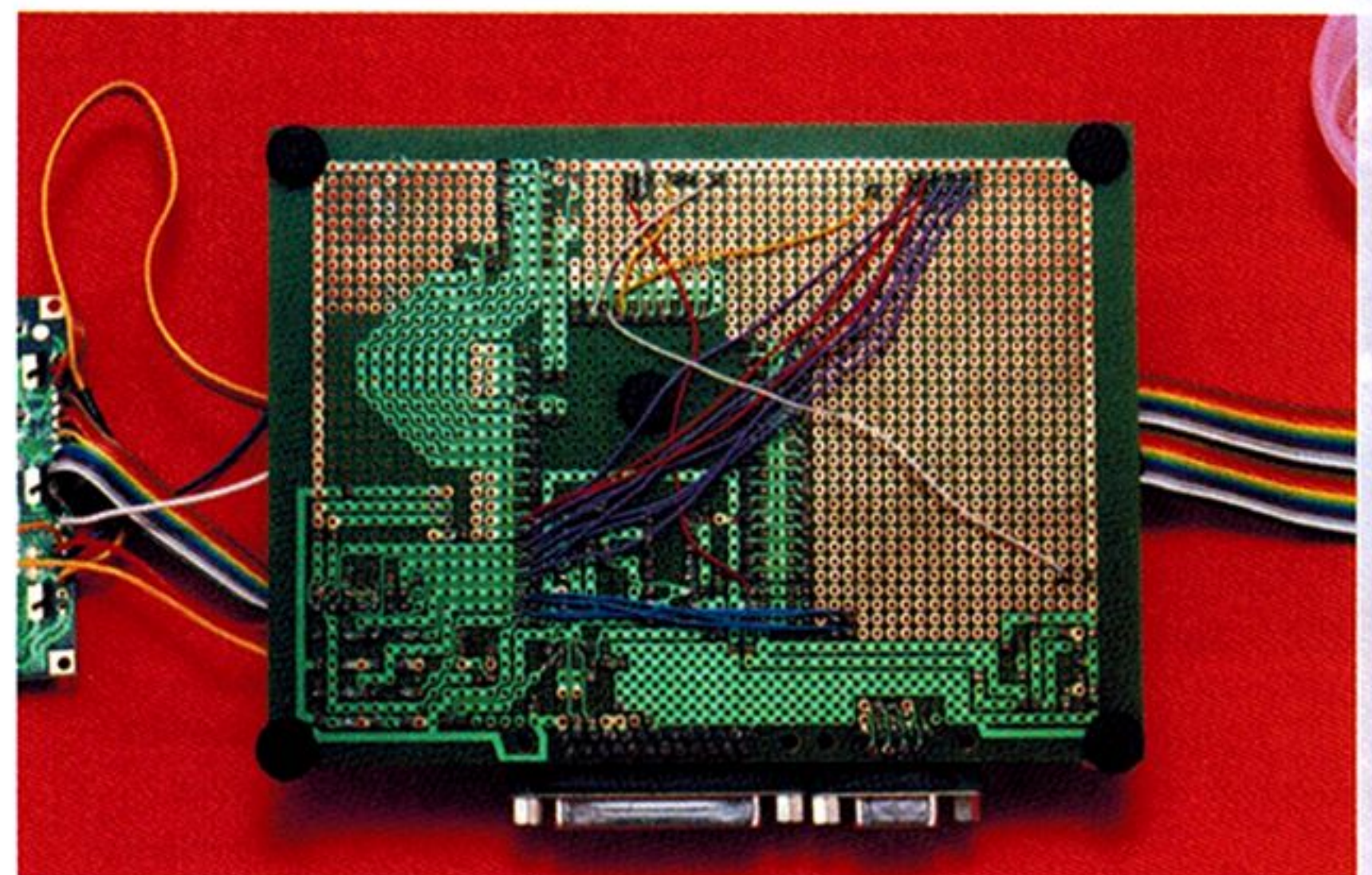



表1-2-a 動作モード別ポート機能一覧(2)

ポート	概要	端子	モード1	モード2	モード3	モード4	モード5	モード6	モード7
ポートA	・8ビットの入出力ポート ・シュミット入力	PA$_7$/TP$_7$/TIOCB$_2$ /A$_{20}$	プログラマブルタイミングパターンコントローラ(TPC)出力端子(TP7)、16ビットインテグレーテッドタイマユニット(ITU)の入出力端子(TIOCB2)と入出力ポートの兼用		アドレス出力端子(A$_{20}$)		TPC出力端子(TP7)、ITUの入出力端子(TIOCB2)と入出力ポートの兼用	アドレス出力端子(A20)	TPC出力端子(TP$_7$)、ITUの入出力端子(TIOCB$_2$)と入出力ポートの兼用
		PA$_6$/TP$_6$/TIOCA$_2$ /A$_{21}$/CS$_4$ PA$_5$/TP$_5$/TIOCB$_1$ /A$_{22}$/CS$_5$ PA$_4$/TP$_4$/TIOCA$_1$ /A$_{23}$/CS$_6$	TPC出力端子(TP$_6$～TP$_4$)、ITUの入出力端子(TIOCA$_2$、TIOCB$_1$、TIOCA$_1$)、CS$_4$～CS$_6$出力端子と入出力ポートの兼用		TPC出力端子(TP$_6$～TP$_4$)、ITUの入出力端子(TIoCA$_2$、TIOCB$_1$、TIOCA$_1$)アドレス出力端子(A$_{23}$～A$_{21}$)、CS$_4$～CS$_6$出力端子と入出力ポートの兼用		TPC出力端子(TP$_6$～TP$_4$)、ITUの入出力端子(TIOCA$_2$、TIOCB$_1$、TIOCA$_1$)、CS$_4$～CS$_6$出力端子と入出力ポートの兼用	TPC出力端子(TP$_6$～TP$_4$)、ITUの入出力端子(TIOCA$_2$、TIOCB$_1$、TIOCA$_1$)、アドレス出力端子(A$_{23}$～A$_{21}$)、CS$_4$～CS$_6$出力端子と入出力ポートの兼用	TPC出力端子(TP$_6$～TP$_4$)、ITUの入出力端子(TIOCA$_2$、TIOCB$_1$、TIOCA$_1$)と入出力ポートの兼用
		PA$_3$/TP$_3$/TIOCB$_0$ /TCLKD PA$_2$/TP$_2$/TIOCA$_0$ TCLKC PA$_1$/TP$_1$/TEND$_1$ /TCLKB PA$_0$/TP0/TEND$_0$ /TCLKA	TPC出力端子(TP$_3$～TP$_0$)、DMAコントローラ(DMAC)の出力端子(TEND$_1$、TEND$_0$)、ITUの入出力端子(TCLKD、TCLKC、TCLKB、TCLKA、TIOCB0、TIOCA0)と入出力ポートの兼用						
ポートB	・8ビットの入出力ポート ・LED駆動可能 ・PB$_3$～PB$_0$はシュミット入力	PB$_7$/TP$_{15}$/DREQ$_1$ /ADTRG	TPC出力端子(TP$_{15}$)、DMACの入力端子(DREQ$_1$)、A/D変換器の外部トリガ入力端子(ADTRG)と入出力ポートの兼用						
		PB$_6$/TP$_{14}$/DREQ$_0$ /CS$_7$	TPC出力端子(TP$_{14}$)、DMACの入力端子(DREQ$_0$)、CS$_7$出力端子と入出力ポートの兼用						TPC出力端子(TP$_{14}$)、DMACの入力端子(DREQ$_0$)と入出力ポートの兼用
		PB$_5$/TP$_{13}$/TOCXB$_4$ PB$_4$/TP$_{12}$/TOCXA$_4$ PB$_3$/TP$_{11}$/TIOCB$_4$ PB$_2$/TP$_{10}$/TIOCA$_4$ PB$_1$/TP$_9$/TIOCB$_3$ PB$_0$/TP$_8$/TIOCA$_3$	TPC出力端子(TP$_{13}$～TP$_8$)、ITUの入出力端子(TOCXB$_4$、TOCXA$_4$、TIOCB$_4$、TIOCA$_4$、TIOCB$_3$、TIOCA3)と8ビットの入出力ポートの兼用						

表2-3 メモリマップされたG1216のアクセス方法

アクセスする領域	動作	H8のインストラクション
左半分	データ書き込み	1) Port Bのビット5を0に設定 2) 0x400001へデータを書き込み
	データ読み出し	1) Port Bのビット5を1に設定 2) 0x400001からデータを読み込み
	命令書き込み	1) Port Bのビット5を0に設定 2) 0x400000へデータを書き込み
	命令(状態)読み出し	1) Port Bのビット5を1に設定 2) 0x400000からデータを読み込み
右半分	データ書き込み	1) Port Bのビット5を0に設定 2) 0x600001へデータを書き込み
	データ読み出し	1) Port Bのビット5を1に設定 2) 0x600001からデータを読み込み
	命令書き込み	1) Port Bのビット5を1に設定 2) 0x600000へデータを書き込み
	命令(状態)読み出し	1) Port Bのビット5を0に設定 2) 0x600000からデータを読み込み

表3-1 ポケットボードの仕様

キーボード	QWERTY式 70キー	
液晶	グラフィック	300x84ドット
	漢字フォント	12x12ドット
外部インタフェイス	デジタル携帯インタフェイス	16ピン
	シリアルインタフェイス	4ピン
電源	電池	単4アルカリ乾電池x2
消費電力	定格電流	60〔mA〕
	消費電力	0.18〔W〕
	電池寿命	26時間(連続使用)
外観仕様	外形寸法	166.2x86x23.8〔mm〕
	重量	199〔g〕

6 (16Mバイト，内部ROM&RAM有効の外部メモリアクセス付き)だと，ひとつの$\overline{CS}$信号あたり，2Mバイトもあり，LCDコントローラだけで4Mバイトも占有してしまうのはもったいないような気もしますが，今回は，2MBの外部メモリ空間よりも1本のI/Oポートピンが貴重なのでCSピンを使用します。

データバス上の値は，データなのか，LCDモジュールへの命令なのかを示す，D/$\overline{I}$という制御信号がありますが，これも$\overline{CS}$信号と同時に確定すればよいようで，アドレスバス0を接続します。つまり，

偶数アドレスへのアクセス(アドレスバス0=0)
データとしてアクセス
奇数アドレスへのアクセス(アドレスバス0=1)
命令としてアクセス

と自動的に割り振ります。

以上の組み合わせをまとめると，表2-3のようになります。

キーボード

CPUがあるだけでは携帯ゲーム機にはなりません。画面と入力装置が必要です。電子工作に比べて機械工作はコストがかかり，0からの製作は難しいので，とりあえず手元にあったポケットボードピュアのキーボードを使用しました。

ポケットボードピュアとは，NTTドコモから発売されている携帯電話用メール端末で，表3-1のような仕様を持っています。東京や大阪で通勤電車に乗れば，実際にポケットボードを使っているOLさんの姿を見かけることでしょう。量販店へ行けば3000～4000円で買えるはずです。

キーボードは表3-1にあるように，単なるマトリクス上にスイッチが集まったものです。H8/3048のPort 4はプルアップ抵抗を内蔵していますので，こちらを入力側に，その他の端子を余ったポート(Port AとPort B)を出力側として端からスキャンしてやればよいでしょう。

具体的な手順は以下のとおりです。

1) Port A, Port Bのすべてのピンを，Low出力にする
2) Port 4には内蔵のプルアップ抵抗が使用されているので，キーが押されていなければ，Port 4の状態はすべてHighになる。キーが押されていれば，1)の出力が回り込んでくるのでLowになる
　→ Port 4の入力データが0xffなら，キー入力はなし
3) Port A，Port Bのうち，ひとつのピンだけが1になるようにする。このとき，Port 4の状態が変化すれば，1にしたピンの列に押されたキーがあるはず
4) Port 4で，Lowになったビットを調べ，キーマトリクスのうち，行を調べる
5) 3と4から，行と列がわかるので，どのキーが押されたかがわかる。そこからASCIIコードを判別する

本当は，オートリピート機能(「A」を押しっぱなしにすると，A，A,AAAAA……と自動的に繰り返される機能)とかNキーロールオーバー機能(「A」を放す前に「N」を押しても，正しく「N」が認識されるようにす

表3-2 ポケットボードのキーボードマトリクス(二重線内)とAKI-H8との接続表

H8	AKI-H8	ポケットボードのピン	PA0 / 1-8	PA1 / 1-9	PA2 / 1-10	PA3 / 1-11	PA4 / 1-12	PA5 / 1-13	PA6 / 1-14	PB0 / 1-16	PB1 / 1-17	PB2 / 1-18	PB3 / 1-19	PB4 / 1-20
			2	3	9	10	11	12	13	14	15	16	18	20
P4 0	1-31	1	シフト											on/off
P4 1	1-32	2		↓	/	+	@	==	2	3	F4	F5	1	
P4 2	1-31	3		→	>	L	P	O	X	C	V	B	Z	
P4 3	1-34	4		←	<	K	O	9	W	4	5c	6	Q	
P4 4	3-3	6		変換	M	J	I	8	S	D	F	G	A	
P4 5	3-4	8		N		H	U	7	E	R	T	Y		
P4 6	3-5	17		↑	*	[	-	全/半	カナ/英	R/ひら	スペース	挿/上		
P4 7	3-6	19		実行	]	削除	後退	¥	F1	F2	F3	F6	戻る	

写真5 ポケットボードのキーボード部分を流用

写真6 LCDパネル裏面の配線例

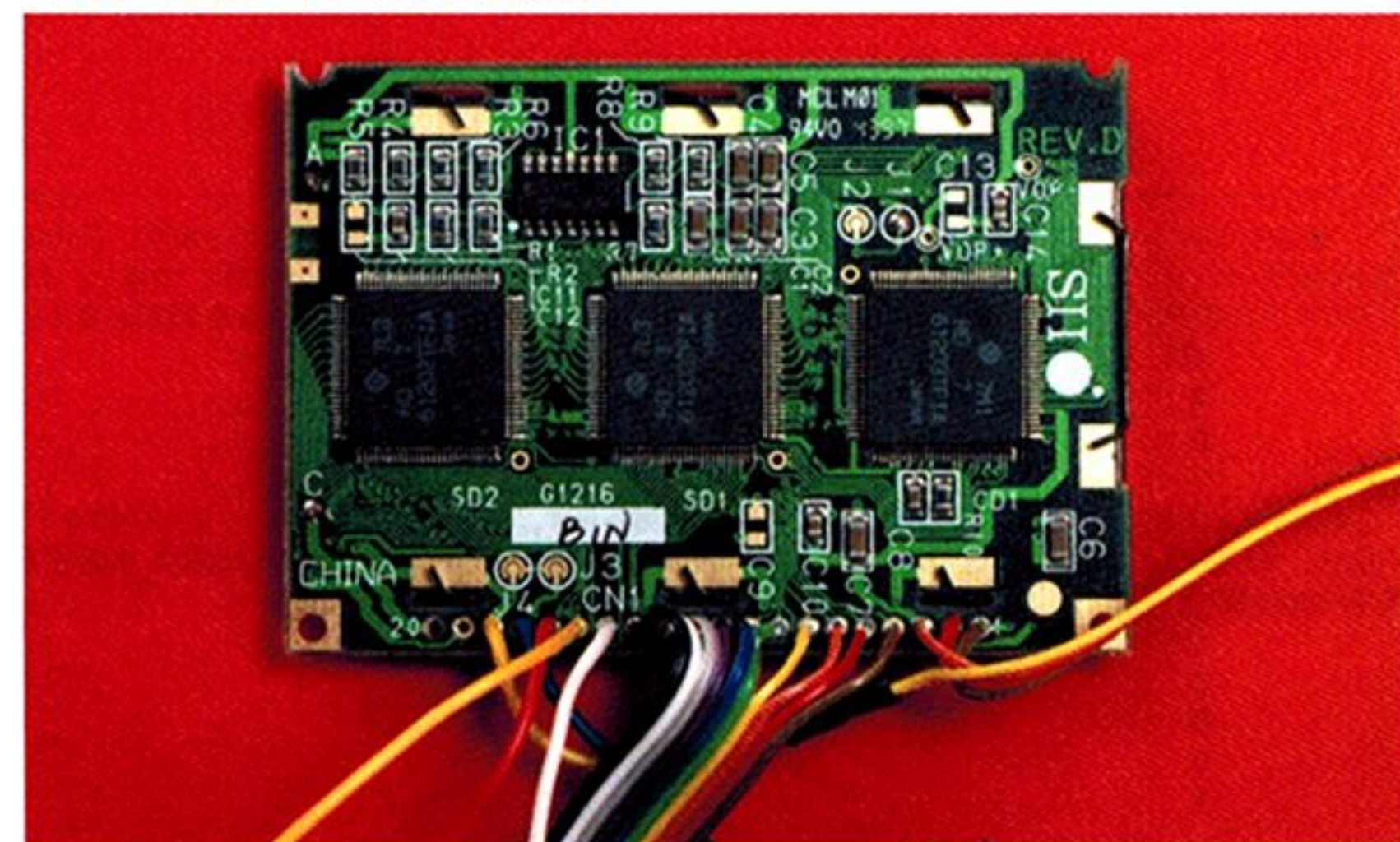



写真7　今回組み上げたシステム

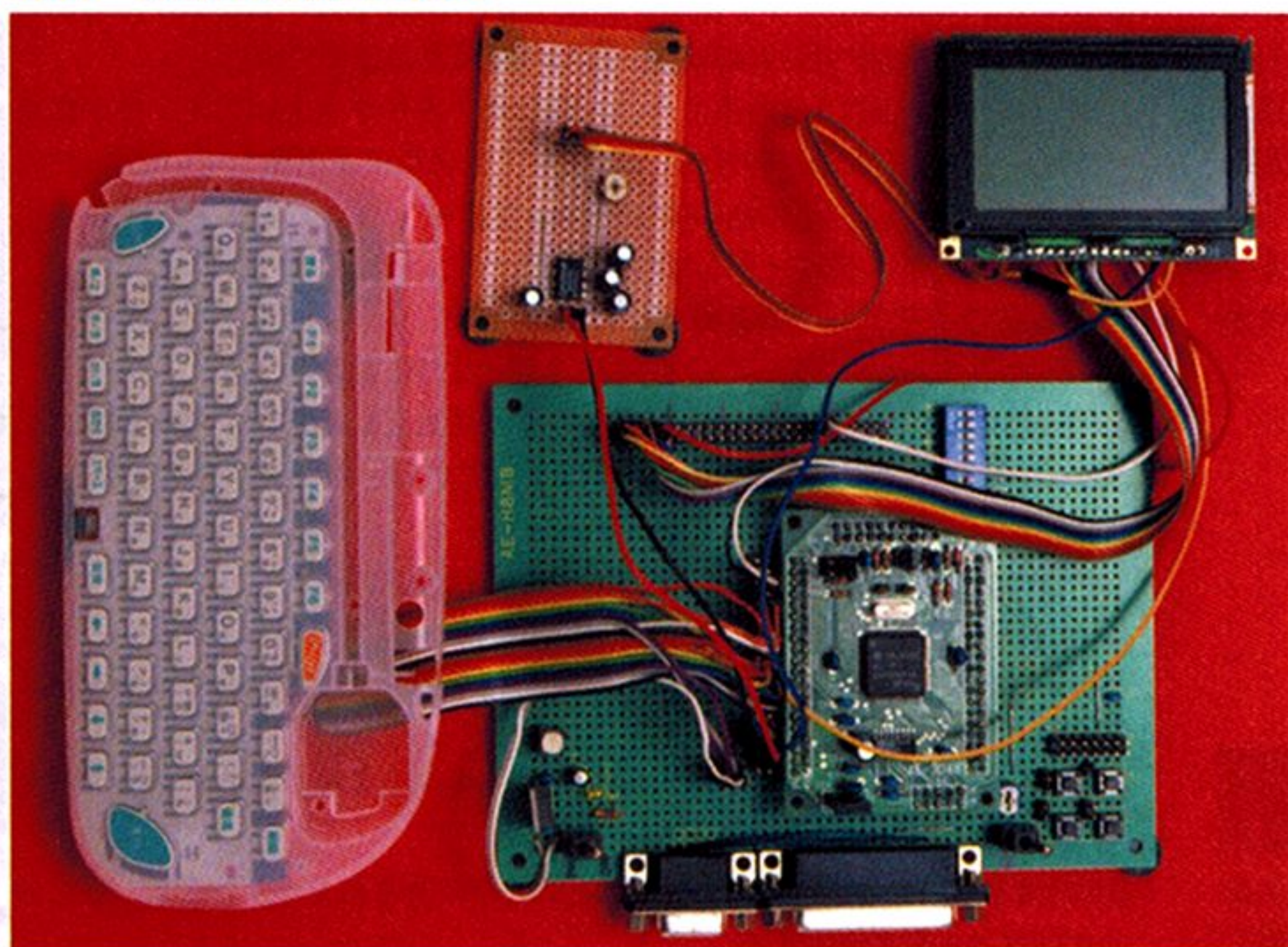
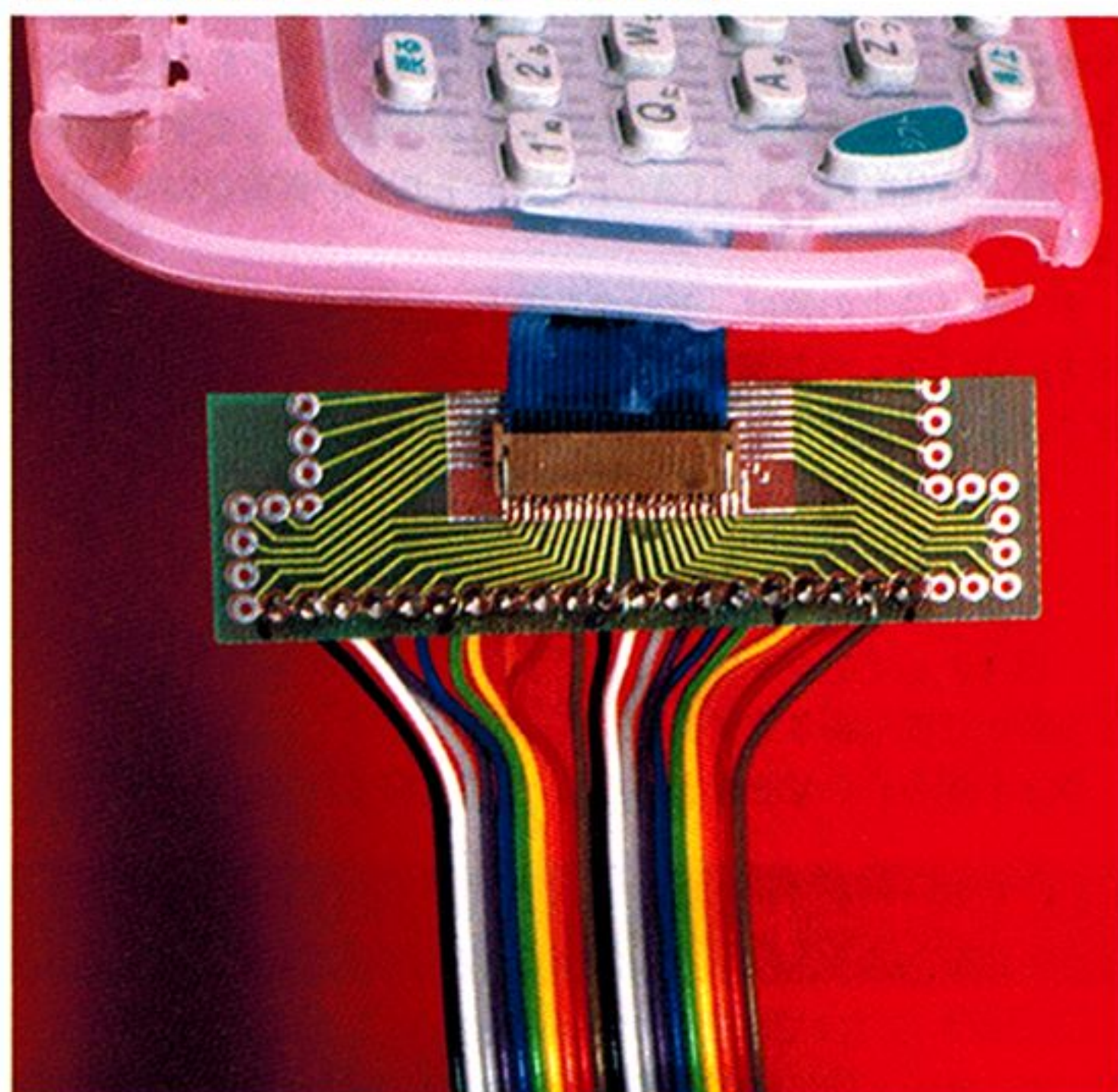



写真8　LCDのフレキシブルケーブルを接続

る機能）などを実現したかったのですが，これらの機能はRTOSの実装が終わり，タスクやデバイスドライバがタイミング生成をどのように行うべきかはっきりした時点で組み込みます。たとえば，オートリピート機能は，あるキーが一定期間押されっぱなしのときに作動し始めますが，RTOSがない場合，この「一定期間」は空ループにより作成します。タイマ割り込みを使用できなくもないのですが，限られたタイマ割り込みは，もう少しほかの機能に回したいものです。RTOS実装システムにおいて，空ループは禁じ手のひとつですから，RTOS完成後にキーボードドライバはちゃんと書き直しましょう。

付属CD-ROMの，

pda/h8/loop

というディレクトリに，キーボードの入力を読み取って，LCD画面（メモリマップされたG1216）に表示するというプログラムを収録しておきましたので参照してください。サンプルプログラムとして極力単純化したので，画面もスクロールしませんし，カーソルキーも無効です。唯一，使える特殊キーはシフトキーのみです。

次回予告

今回はマイコンにLCDモジュール，キーボードを繋げるだけで，手一杯でしたが，次回からは簡単なRTOSを作ったりして，もう少しシステムとしてまとめていきます。システムとして，各ペリフェラルはどのようにタイミングを取るべきか決まれば，簡単な音楽機能も入れたいですね。

また，せっかく外部メモリ空間をアクセスできるように，ポートをやりくりしたのでフラッシュメモリなどの外部メモリ接続も検討します。

そういえば，PDAといえばタッチパネルなのですが，なかなか情報を入手できません。もし読者の方でなにかご存じの方がいらっしゃいましたら，編集部までお知らせください。

＊　＊　＊

と，ここまで書いて，320×200ドットのLCDモジュールを発見しました。え，クレジットカードが使えない？　国際為替で支払うの？　今は黄金週間中，郵便局は閉まっている，あわわ締め切りが……待て，次号（編注：そういう時期に書かれた原稿です）。

参考文献

H8の扱い方およびモニタプログラムのモディファイ方法に関して。

1-1）日立製作所「H8/3048シリーズh8/3048F-ZTATハードウェアマニュアル」平成11年3月
1-2）日立製作所「H8/300Hシリーズプログラミングマニュアル」
1-3）「H8/300H組み込み型モニタのフルカスタマイズ方法」（モニタプログラムのダウンロードページから，ダウンロードできます。）
1-4）トランジスタ技術編集部「AKI-H8マイコン・ボードにモニタを移植する・関連プログラムとデータなど（改訂 第1版）」http://www.cqpub.co.jp/toragi/DLF/TR9809M1.htm

LCDの原理，およびモジュールの扱いに関して。

2-1）真野　智秀「LCDディスプレイの表示原理と駆動方法」トランジスタ技術2000年3月号
2-2）Beyond the river「H8/3002，H8S/2144 CPUボード用液晶コントロールボード YH08-13 取扱説明書」　1999.3.26
2-3）Seiko Instruments「G1213，1216 Modules with built-in Data RAM」Data Sheet
2-4）HITACHI「HD61202U Dot Matrix Crystal Graphic Display Column Driver」Data sheet
2-5）Maxim「MAX680 / MAX681 +5V to ±10V Voltage Converter」Data Sheet

＊2：G1216はインタフェイス用のICとして，日立製作所のHD61202Uを使用しています。このICはCPUとのインタフェイスにモトローラ方式（といっていいのか？）を採用しています。つまり，$\overline{CS}$でデバイスが選択されたことを知り，R/$\overline{W}$でそのアクセスが読み出しか書き込みかを判別します。

$\overline{CS}$	R/$\overline{W}$	
0	0	書き込み
0	1	読み出し
1	x	なにも起こらず

これに対し，H8/3048は同じ日立製作所製であるに関わらず，インテル方式（といっていいのか？）を採用しています。これは$\overline{RD}$信号で読み出し，$\overline{WR}$信号で書き込みと，アクセスの種類を判断する信号が2種類存在します。

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	
0	0	0	禁止
0	0	1	書き込み
0	1	0	読み出し
1	x	x	なにも起こらず

実際には，H8は16ビットCPUで，8ビットのメモリアクセスをサポートしていますから，$\overline{WR}$信号ではなく$\overline{HWR}$信号（上位バイト書き込み）と$\overline{LWR}$（下位バイト書き込み）という2つの信号に分かれています。今回は8ビット外部アクセスしか用いていないので，$\overline{HWR}$信号＝$\overline{WR}$信号とみなせます。

それはともかく，この2つのアクセス方式を見ると，R/$\overline{W}$信号と$\overline{WR}$信号をつなげば辻褄があることがわかりますね。

＊＊3：本当は，ポケットボードについている液晶パネルが，軽いし，コントラストもいいし，解像度も高いし，ドットピッチも細かいし，おそらく3.3〔V〕ロジックレベルだろうしというわけで，ぜひとも使いたかったのですが，どうも使い方がわからず，今回は市販のモジュールを使用しました。最初は，解析も簡単に行くと思ったのですが，なかなか難しく，3台もポケットボードをダメにしました。17〔V〕は流れているわ，40〔MHz〕近いクロックは走っているわ，とても単3×2本で動くオモチャとは思えません。ちょっと隣のピンとショートさせると，しゅわぁぁ～んといって，静かにお亡くなりになります。いやぁ，毎週，青いポケットボードを買いにくる私をドコモショップのおねーさんはなんだと思っただろうなぁ。

部品の購入先

今回は，近所の電子パーツ屋へ行っても購入できそうにないものをかなり使用しているので，以下に購入先をまとめておきました。地方在住の方，インターネットを初めて間もない方は参考にしてください。

■CPUボード(AKI-H8)

本文中でも述べたとおり，秋葉原の秋月電子通商(http://www1.tomakomai.or.jp/akizuki/index.html)で購入しました。ボード単体(3800円)での発売も行っているのですが，最初の1枚目は「AKI-H8開発キット(即使えるキット)」(7800円)を買っておきましょう。AKI-H8＋アセンブラ・リンカ＋Flash書き込みプログラム＋専用マザーボード(H8/3048F内蔵フラッシュメモリの書き込み機能付き)＋DC19(V)スイッチング電源，のお得なセットになっています。

注意1：ソフトウェアの媒体が，1.2(MB)のフォーマットの3.5インチフロッピーなので，対応したドライブも持っていない場合は，別売りのCD-R(500円)を同時に購入します。

注意2：付属のスイッチング電源は部品剥き出しの基板です。AC100(V)を入れる装置なので，どんなに間違っても内部を触れないようにケースに入れなければいけません。ケース加工が苦手な人は，別売りの15(V)　ACアダプタ(850円)を購入したほうがよいでしょう。なお，通常のACアダプタ(外付けモデムやコードレス電話などに付属するもの)は，フラッシュメモリ書き込みに使用できるほど安定化されていません。手持ちのACアダプタを流用して，あとでトラブルに悩まされるよりは，実績のあるものをAKI-H8と一緒に購入することをおすすめします。

■LCDモジュール

Optrex社や日立製のLCDモジュールは秋葉原でもときどき見かけますが，セイコーインスツルメンツのLCDモジュールはスポット的にしか見かけません。なかなか一定して置いている店はないようです。

お店ではないですが，イエローソフト(http://www.yellowsoft.com)が自社製のH8/3002，H8S/2144CPUボードのオプションツールとして，

YH08-13用液晶パネル　LCD-1

という商品名で扱っています(9600円)。ホームページには載ってないですが，トラ技などの雑誌広告に掲載されています。実際には，G1216に20ピンのフラットケーブルが付いたものでした。

日本ではなく米国のセイコーインスツルメント社のホームページへ行き(http://www.seiko-usa-ecd.com/)，LCDモジュールを探すと，サンプル要求ボタンのあるものがいくつかあります。実際には，セイコーインスツルメンツ社ではなく，Mouser Electronics(http://www.mouser.com)という通販店へのリンクになっているのですが，ここは企業だけでなく，個人相手の販売も行うとのことでした。今回のG1216も1個からの販売もOKとのことです。4月下旬現在，1個36ドルでした(ただし，日本からの注文は，合計100ドル以上のみ対応可能とのこと)。

日本製品なのに米国から輸入するというのがもったいないですが，電子部品が複雑/多様化する昨今，このような通販店を利用する機会は今後，増加しそうです。

クレジットカードを持っていれば，Webページ上で注文を行うことは可能なのですが，まぁ，いろいろあったりして，実際，何回かは電話をかけなければなりませんでした。ちなみに，この電話は米国のフリーダイアル(800番)なので，KDDなどからは接続できずAT&TやMCIのコーリングカード経由でダイアルします。商品が特殊なので通関で引っかかった場合，運が悪いと送り元の協力が必要かもしれません(軍事目的に転用できないことなどの証明)。少し敷居が高いのは事実ですが，海外通販の経験が何回かあれば，こちらを利用するのもよいでしょう。

■ポケットボードピュア

近所のドコモショップ，家電品量販店で購入できます。ドコモの携帯電話を持っていなくても購入できるようです。本当はもっといろいろな部品を流用したかったのですが，今回はキーボードしか流用できませんでした。*3

(編注：現状では同一製品は入手困難か)

■MAX680

秋葉原の若松通商(http://www.wakamatsu.co.jp)で購入しました(590円)。本文中にあるようにMAX681が入手できれば，外付けコンデンサが不要な分便利でしょう。

■フレキシブルケーブル用コネクタ

回路図には載っていませんが，AKI-H8とキーボードの間にコネクタがあります。ポケットボードピュアのキーボード接続に使用されているフレキシブルケーブルは半田ごてを当てると溶けてしまうので，直接ワイヤをつなげることができません(はい，私も1個ダメにしました)。購入できる方は0.8(mm)ピッチの20pin用コネクタを購入してください。あてのない人はポケットボードピュアの基板から剥がします。

コツは，もうこれ以上乗せられないというくらいコネクタにハンダを盛ることです。もちろんピン間はショートさせます。ひとつのハンダ玉で，ピンの半分くらい繋げたというところでハンダごてを外し，冷めないうちにカッターナイフの逆面をソケットの裏側に差し込んで軽く数ミリ持ち上げます。次に同じ作業を残り半分に行います。この動作を数回繰り返すと，ポロッとコネクタが外れます。

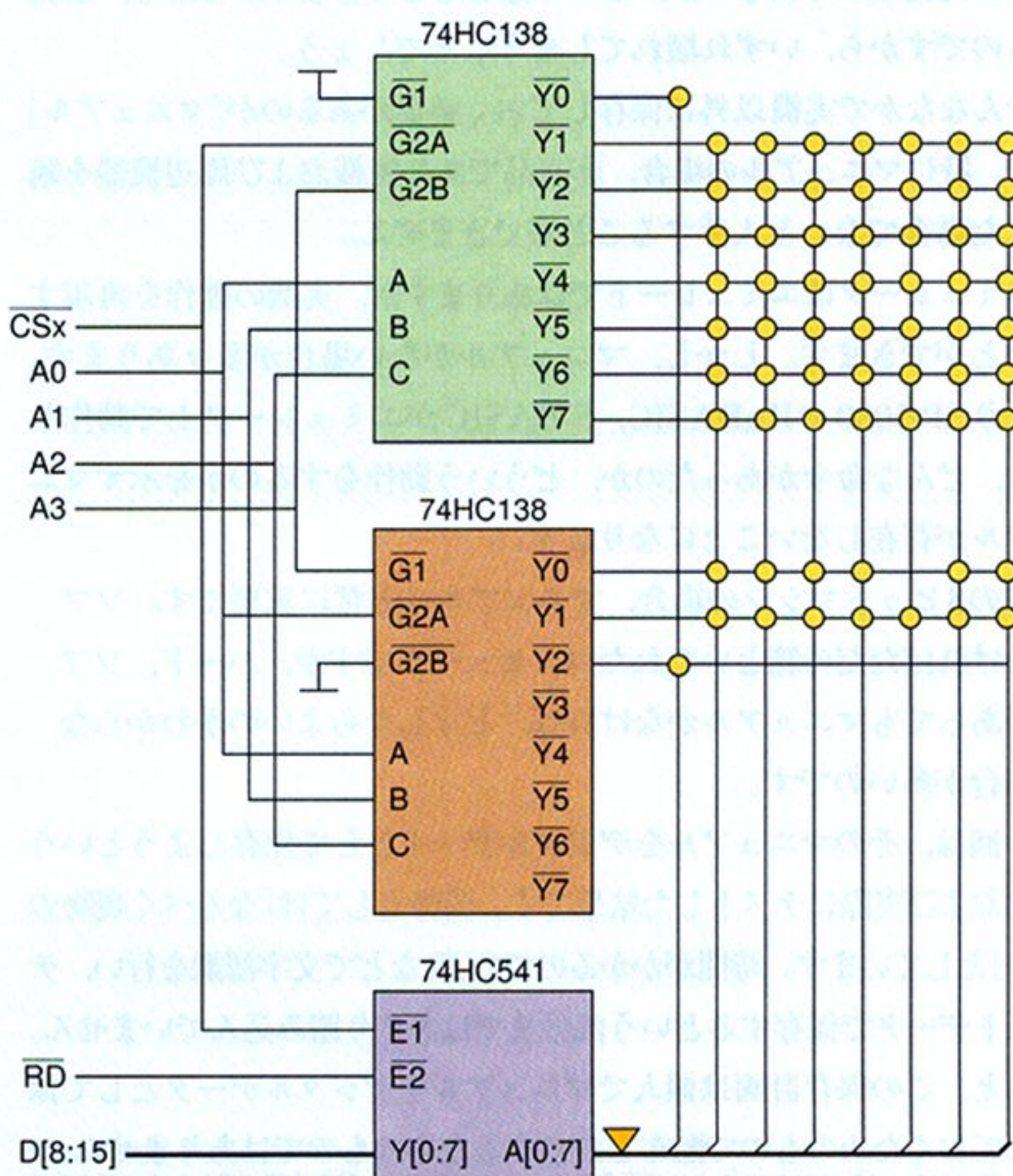
また，写真8にあるようにコネクタは直接マザーボードにハンダづけせず，サンハヤト社の0.8(mm)QFPピッチ変換基板(ICB-012)を半分に切ったものを使用し，そこから20pinフラットケーブルを用いてマザーボードと接続しています。

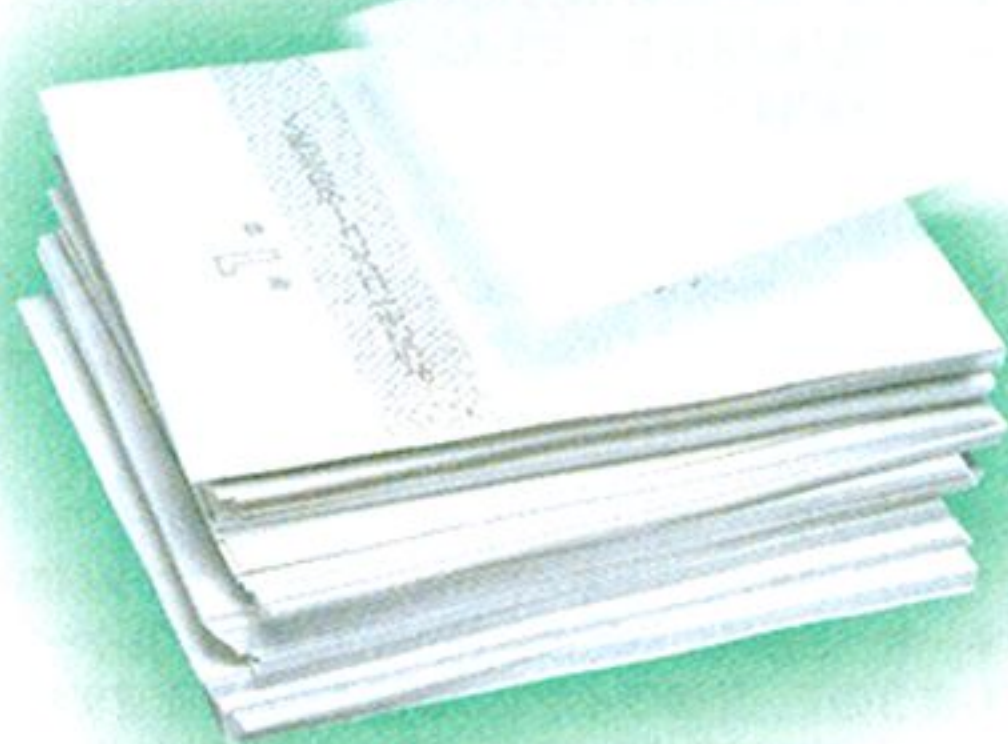
もう一つのキーボードのスキャン法

本文中で，ポートがないと繰り返しておきながら，キーボードのスキャンに3ポートも使用しています。これは，単に外付け部品をつけるのが嫌だったからなのですが，以下のように接続すれば，まったくポートを使用せずにメモリマップドデバイスとしてアクセスできるようになります。74HC138で下位アドレス4ビットをデコードし，74HC541で対応するマトリクスのスイッチの状態を取り込みます。GALと呼ばれるデバイスを用いれば，2つの74HC138が1個にまとまらないこともないのですが，消費電力という観点やケーブルのコネクタ脱着時に発生する静電気が回り込んだときに74HCxxとGALでどちらが耐性が高いかという観点から，74HC138のほうが有利に思えます。

消費電力も74HC138，74HC541ともに0.08〔mA〕だそうですので，ほとんど電池の負担にはならないでしょう。

将来，ポートがさらに足りなくなってきたら，こちらのほうに移行するかもしれません。

特別企画

MZシリーズ マニュアル保存計画

古籏一浩
Furuhata Kazuhiro

次第に過去のものとなりつつある旧型パソコン。しかし時代のなかで光る個性を見せているそれらのマシンは，最近の互換機とは違ったユニークな存在です。それらの機能はエミュレータで残すことができても，問題になってくるのがマニュアルなどの関連情報でしょう。ここでは個人個人でマニュアルを残していくため，そういったドキュメントをスキャンしてまとめる手順を紹介します。

MZは遠くなり……

MZシリーズが発売されてから，すでに20年近く経過しました。いまではエミュレータ上で動く過去のものとなっています。でも，手元にあるMZは動きます。頑丈です。このまま，結構年月が経過しても持ちこたえるのではないか，という感じもしてきます。しかし，形あるものですから，いずれ壊れてしまうことでしょう。

そんななかで実機以外に保存しておく必要があるのが「マニュアル」です。特にマニュアルの場合，非売品であり実機および周辺機器を購入した場合でないと入手することができません。

エミュレータはエミュレートではありますが，実機の動作を再現することができます。しかし，マニュアルがない場合が多々あります。つまりSP-5030やHuBASIC，S-BASICがエミュレータ上で動作しても，どんな命令があったのか，どういう動作をするのかを示すマニュアルが存在しないことになります。

昔の8ビットマシンの場合，マニュアルは非常に重要です。ソフトなければただの箱といわれたコンピュータですが，ハード，ソフトがあってもマニュアルがなければ，どうしたらよいのかわからない場合が多いのです。

今回は，そのマニュアルをデジタルデータとして保存しようという計画および実際にテストした結果です。姿勢としては「なるべく現物のまま」としています。時間がかかるのでOCRなどで文字認識を行い，テキストデータで保存するという部分まではあまり踏み込んでいません。

あと，この保存計画は個人でマニュアルをデジタルデータとして保存しておくためのもので配布したりするためのものではありません。[*1]

*1　希望としてはCD-ROM，Webなどで配布したいところですが。配布許可が得られるかどうかはシャープの懐の深さ次第でしょうか。

MZシリーズのマニュアル

さてMZシリーズのマニュアルですが，周辺機器などにもマニュアルが付属しており，実際のところどんなマニュアルが存在したのかが明確に把握できていません。現段階では**表1**のような状態です。「こういうマニュアルも存在する」というのがありましたら，最後に記述してあるメールアドレスまでメールをいただければ幸いです。

また，今回対象としているMZは80K～1500，80B～2861，3500～6550までであり，AXなどは対象外です。

マニュアルは前にも書いたように「なるべく現物のまま」保存するためスキャニングしPDF（Portable Document File）形式としました。PDF形式にしたのは複数のページ画像を1ファイルに集約できることと，圧縮方法が選択できること，また公開された形式であるという点です。

必要な機材およびソフト

必要なハードウェアはパソコン本体とスキャナ，そしてCD-Rです。今回使用したハードウェアは以下のとおりです。

- PowerMac 8500/132（パソコン）
- GT-7000（スキャナ）
- CD-RWドライブ

ハードウェア以外にソフトウェアも必要になります。使用したソフトウェアは以下のとおりです。

◆Adobe Photoshop 5.0J
◆Adobe Acrobat 3.0J

特に高いPhotoshopを買わなくてもPhotoshop 5.0LE版でも問題ありません。Acrobatは現在4.0ですが3.0でも問題ありません。また，OCRソフトは「読んでココ Ver 6」を使用しました。このソフトはスキャナから読み込みテキストエリアと画像エリアを指定してPDF化してくれるという便利なシロモノです。直接スキャナから読み込まなくても，既存の画像から同様にテキストエリアと画像エリアを分けて保存することができます。

ハード，ソフトともに，あまり高い障壁ではありません。最大の障壁は「気力」です。なんといっても200ページ近くか，それ以上のマニュアルを全ページスキャンし，画像補正などを行うわけですから結構な気力が必要です。気力の次に時間も必要です。よほど時間的余裕がないと難しいといえます。どのくらい時間がかかるかについては後述します。

具体的な手順

まずPhotoshopからスキャニングを行います。レーザープリンタなどで印刷した場合にも耐えられるように取り込み解像度は300dpiとし，グレースケール256階調とします。MZシリーズのマニュアルの表紙，裏表紙はカラーになっているので，この場合はフルカラーで取り込みを行います(図1，図2)。

マニュアルが数冊ある場合は，ページごとに分解して取り込むほうが綺麗です。しかし，いまとなってはマニュアル自体が貴重品であるため分解するのは気が引けてしまいます。ここはPhotoshopで加工して処理するとして単純にページを開いてスキャンを行います。

当然，傾いたり折りの部分の影ができてしまいます。しかし，これに構わずスキャンを行います。スキャンされた画像は図3のような感じになります。

ぱっと見て次のページ(前ページ)の内容まで，ごていねいにスキャンされてしまっています。いわゆる「裏写り」の状態です。これは以下のようにすることで簡単に消すことができます。

・メニューから「色調補正」→「レベル補正」を選択
・図4の△を左に移動

これで裏写りが消えます。また，文字は本当は黒でなければならないのですが，スキャン状態によっては灰色になってしまいます。文字を黒くするには以下のようにします。

・メニューから「色調補正」→「レベル補正」を選択
・図5の▲を右に移動

これで文字が黒くなり鮮明になります。先ほどの裏写り消去と文字を黒くするのは同一画面なので，数秒程度で処理することができるでしょう。また，裏写りは黒い紙をはさむことによりかなり軽減することができます(図6参照)。

ここで，このレベル補正の修正値を覚えておきます。図7の矢印で示している2カ所です。以後，スキャンした画像に対して，この数値を入力するだけで補正を行うことができます。また，「アクション」と

【表1】MZシリーズのマニュアル一覧

機種名	マニュアル名
■MZ-80K ■MZ-80K2 ■MZ-80K2E ■MZ-80C	取扱説明書 BASIC解説 DISK BASIC Double Precision SP-6110 DISK BASIC INTERPRETER PASCAL MACHINE LANGUAGE SYSTEM PROGRAM Z80 PROGRAMMING MANUAL SYSTEM PROGRAM BACK UP FDOS BASIC Compiler
■MZ-1200	OWNER'S MANUAL BASIC MANUAL
■MZ-700	OWNER'S MANUAL DISK BASICマニュアル SYSTEM PROGRAM MANUAL MACHINE LANGUAGE MANUAL
■MZ-1500	初めてお使いになる人のために OWNER'S MANUAL BASIC LANGUAGE MANUAL UTILITIES/APPLICATION MANUAL SYSTEM PROGRAM MANUAL MACHINE LANGUAGE MANUAL
■MZ-80B	DISK BASIC MANUAL ADPP SYSTEM
■MZ-80B2 ■MZ-2000	OWNER'S MANUAL BASIC/MONITOR MANUAL BASIC解説 COLOR BASIC MANUAL DISK BASIC MANUAL DOUBLE PRECISION TAPE BASIC MANUAL PASCAL MANUAL 16-BIT BOARD KIT OWNER'S MANUAL [QD] BASIC LANGUAGE MANUAL FLOPPY DOS MANUAL Z80 PROGRAMMING MANUAL
■MZ-2200 ■MZ-2500	OWNER'S MANUAL BASIC-M25 マニュアル BASIC-S25 マニュアル テレホンソフトマニュアル 初めてお使いになる人のために パソコンファクス25
■MZ-2531	OWNER'S MANUAL V2
■MZ-2861	OWNER'S MANUAL MS-DOS V3.1 マニュアル BASIC-M28 マニュアル 日本語ワードプロセッサマニュアル(入門編) 日本語ワードプロセッサマニュアル(解説編)
■MZ-3500	OWNER'S MANUAL BASIC LANGUAGE MANUAL BASIC LANGUAGE MANUAL[付録]
■MZ-5500 ■MZ-6500	OWNER'S MANUAL [漢字]CP/M-86 MANUAL MS-DOS USERS GUIDE MANUAL BASIC-3 LANGUAGE MANUAL BASIC-3 REFERENCE MANUAL 日本語ワードプロセッサ マニュアル
■MZ-6500C	[漢字]CP/M-86 MANUAL MS-DOS USERS GUIDE MANUAL
◆MZ-80DU	COLOR CONTROL
◆MZ-80SFD ◆MZ-80SD1 ◆MZ-8BIO3 ◆MZ-8BIO4 ◆MZ-1C26 ◆MZ-1E08 ◆MZ-1E18 ◆MZ-1E26 ◆MZ-1E36 ◆MZ-1F07 ◆MZ-1M08 ◆MZ-1M11 ◆MZ-1M12 ◆MZ-1P07 ◆MZ-1R13 ◆MZ-1U03 ◆MZ-1U08	

して登録しファンクションキーを押すだけで処理するようにすればさらに処理が簡単になります。

次に傾きを修正しなければなりません。傾きは以下のようにします。

・メニューから「イメージ」「画像回転」「角度入力」
・0～1.5度範囲で数値を入力

角度は0.4，0.7など小数値が使用できます。だいたい，同じような角度を入力するだけで用は足ります。手動で回転させるとかえって時間がかかってしまいますので角度を入力するほうがよいでしょう。

次に四辺の影を消去します。影の消去は矩形範囲選択ツールを使って消したい部分を囲んでdeleteキーを押すだけですみます。また，若干のゴミなども矩形範囲選択ツールで囲んで消去しておきます。

■ファイルの保存

できあがったら保存しなければなりません。ファイルメニューから保存を選択しPDF形式で保存します。圧縮形式はJPEGでなくZIPを選択します。JPEGでは劣化してしまいますので表紙もZIPで保存します。表紙のサイズが大きくなってしまいますが，それでも6MB程度で許せる範囲です。

ちなみに，B4サイズのマニュアルであれば1ページあたり平均して500KB，A5サイズであれば200KB程度になります。

Acrobatで連結し完成

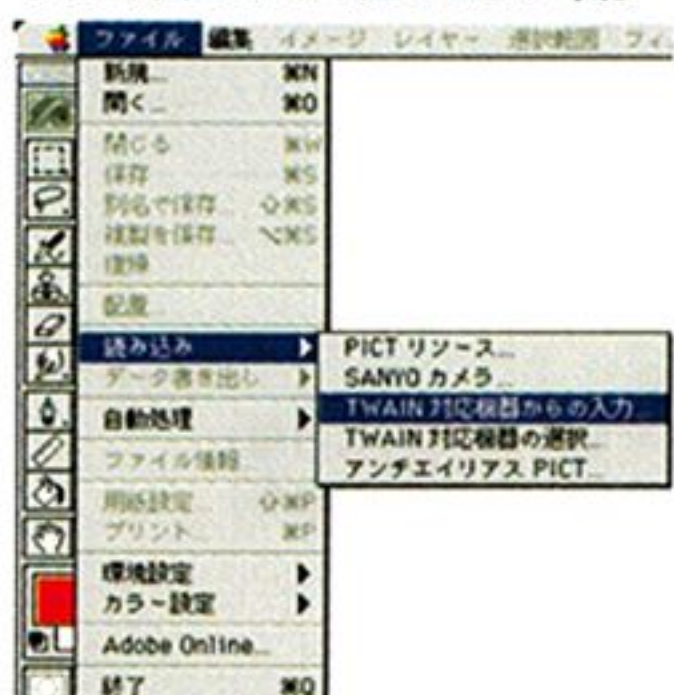
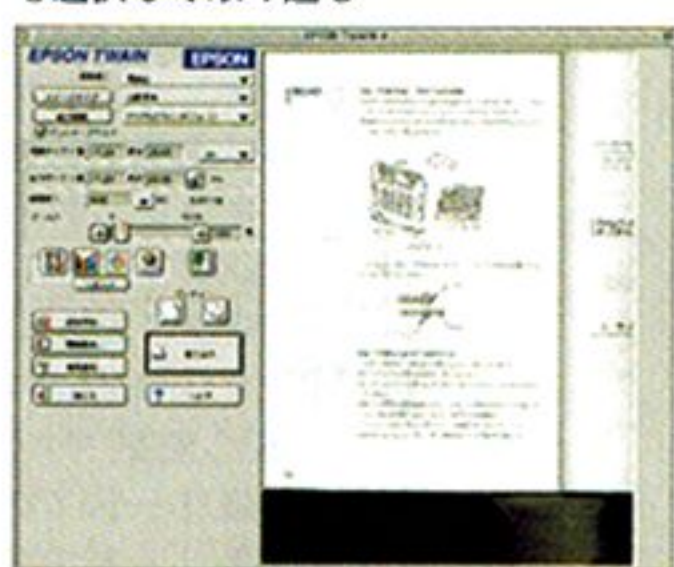
すべてのスキャンが終了し保存したら今度はAdobe Acrobatを使ってページを連結します。まず表紙をコピーし複製を作ります。コピーした表紙をAcrobatで開きます。連結するには以下のようにします。

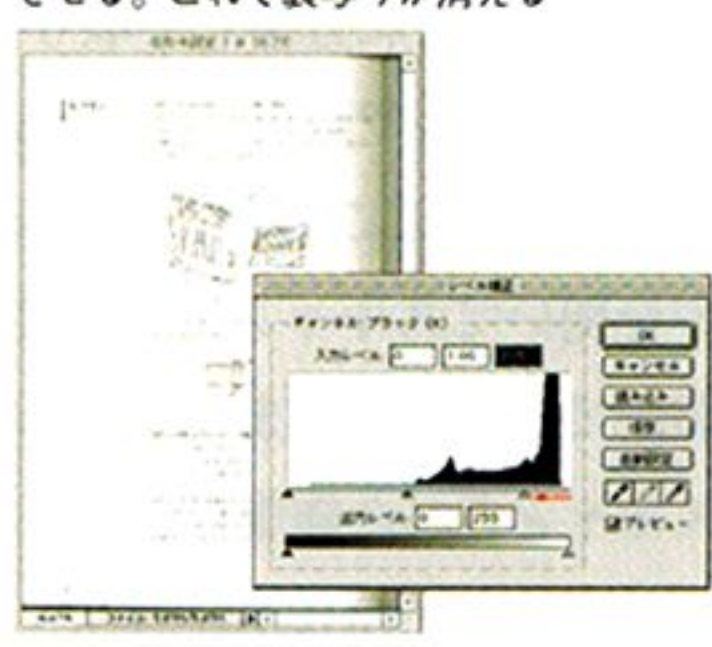
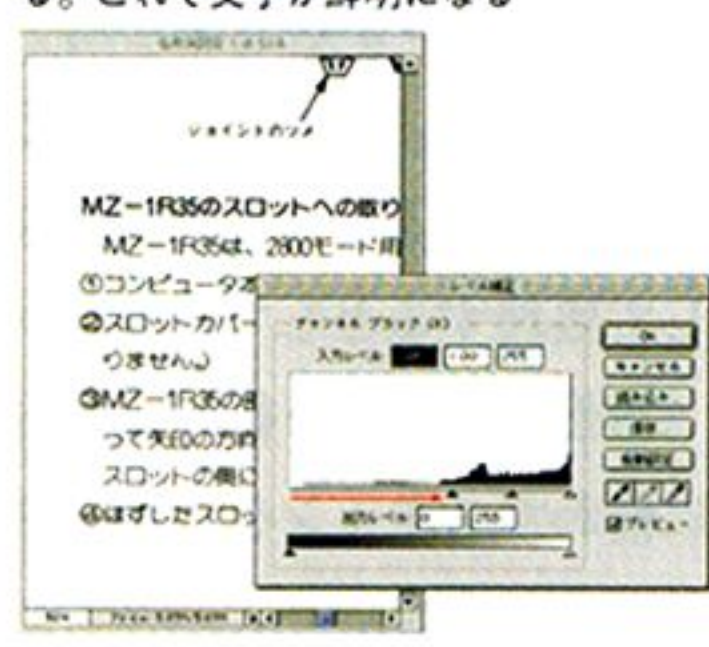
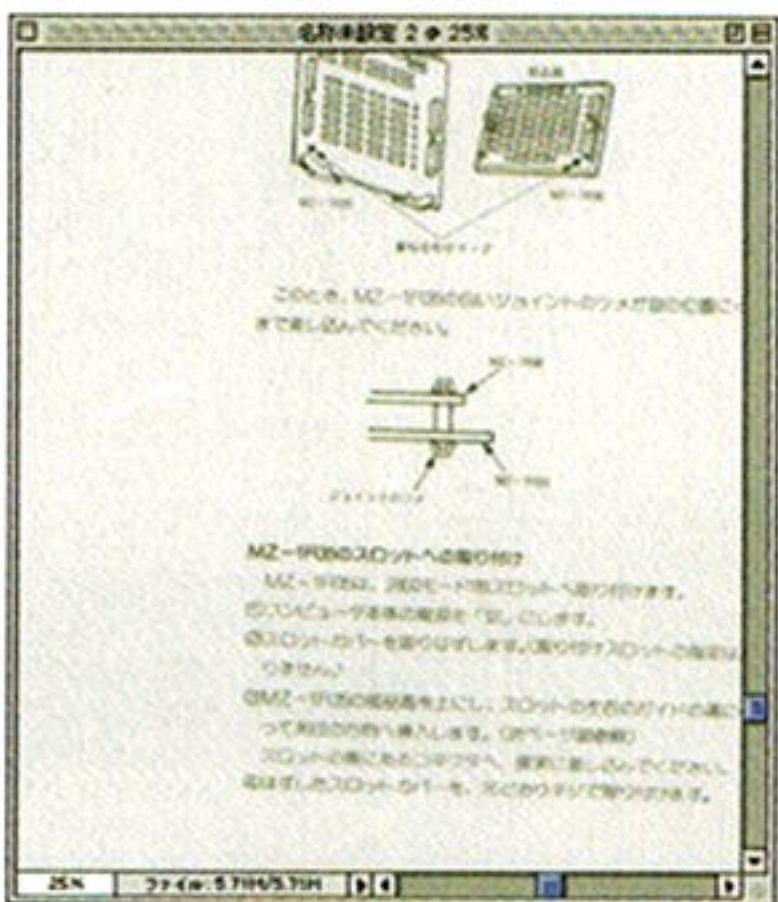
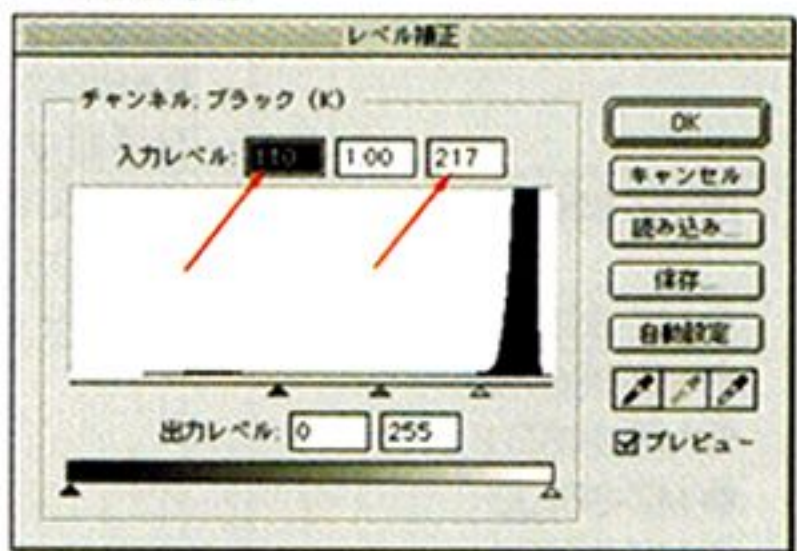
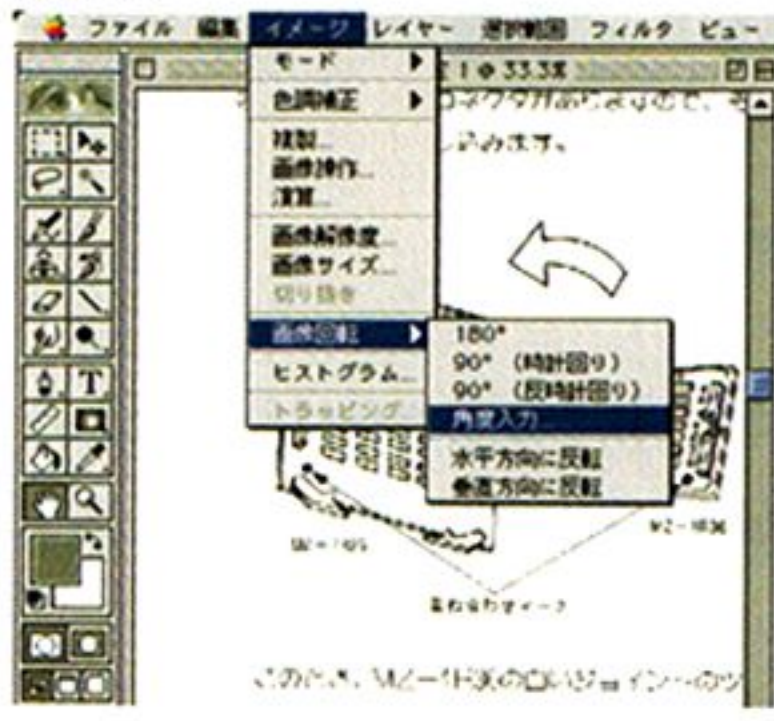
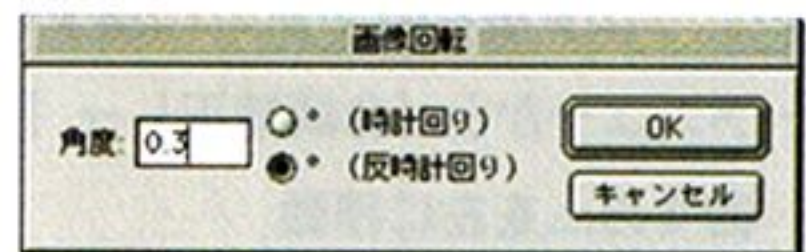
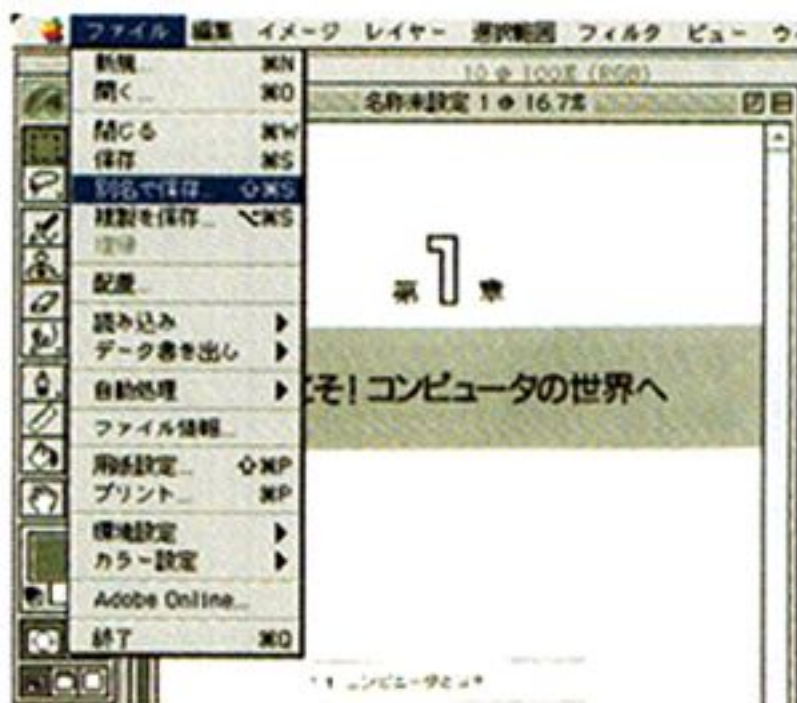
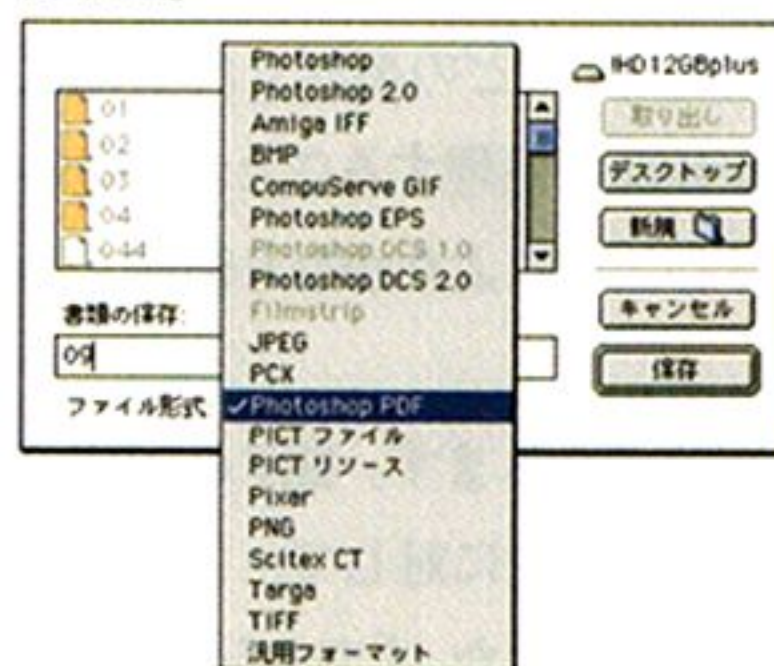
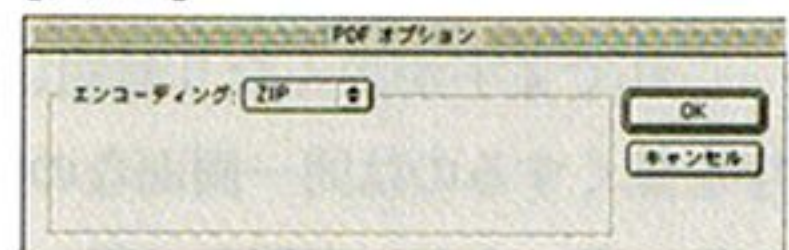
・メニューから「書類」「ページを挿入...」を選択
・連結ファイルを選択
・ダイアログが表示されるので，そのままOK

あとは，この作業を繰り返すだけです。ほかに便利なツールとかがあれば，もっと作業を軽減することが可能です。

また取り込み解像度が300dpiなので，あとでOCRソフトなどで文字を抽出したりすることも可能です。実際に印刷してみると，若干線が太く感じられるものの，違和感のない状態でした。ただし，元のまま保存することを前提にしているため通常の閲覧を行うには，かなりのマシンパワーが必要になってしまいます。閲覧するだけならば120dpi程度でも十分です。

今回行ったのはMZ-2861オーナーズマニュアルとMZ-700 DISK BASICの2つです。PDF後の容量ですが，MZ-2861オーナーズマニュアルが92.7MB，MZ-700 DISK BASICが25.1MBとなりました。

【図1】
PhotoshopのTWAINプラグインを使って取り込む。EPSCANなどを使えばもっと効率よく取り込むことが可能

【図2】
取り込み解像度，カラーかモノクロかを選択して取り込む

【図3】
取り込み直後の画像

【図4】
右側にある△のスライダを少し左に移動させる。これで裏写りが消える

【図5】
左側にある▲のスライダを右に移動させる。これで文字が鮮明になる

【図6】
黒い紙をはさんでから取り込むと裏写りが軽減される。若干暗くなるがレベル補正で明るくすることができるので問題はない

【図7】
矢印の2箇所の数値を覚えておき次回からは直接入力する。アクションで設定すればワンキーで処理可能

【図8】

【図9】
メニューから回転，角度入力を選択し角度を入力する

【図10】

【図11】

【図12】
ファイルメニューから保存を選択。保存形式をPDFにする。保存オプションは画像の圧縮形式をJPEGかZIPかを選択できる。劣化させないためにはZIPを選択する

OCRソフトを使ってPDF化する

とりあえず300dpiの高解像度で取り込んでおけば，あとはどうにでもなります。さすがに全部画像ではPDFといえども画像サイズが大きくなります。やはり画像と文字に分離してPDF化するのがサイズも小さくなり閲覧時も高速に表示できます。

「読んでココ ver.6」では取り込んだ画像からテキストと画像を分離し自動的にPDF化してくれます。取り込み済みの画像も読み込んでPDF化できます(図20)。

PDF化された場合，若干元のマニュアルとは文字形状や書体が異なってしまいますので，元のマニュアルに近い状態で保存しておくには画像のままにしておくほうが賢明です。OCRソフトで作成したPDFはデータサイズがかなり小さくなります。ここまで小さくなればCD-ROM1枚に，ほとんどのMZのマニュアルが入ってしまうと思います。

最後に

MZのマニュアルなんか取っておいて，どうするの？　という人もいるでしょう。いや，大方の意見としては意味のないこと，といったところでしょう。しかし，過去の遺産であれ資料として保存しておくべきではないでしょうか。歴史の変遷というのもありますし，ハードウェア設計やBASICなどソフトウェアの変遷も，マニュアルから読み取ることができます。

すべてが電子化されているのであれば，このような作業自体不要ですが，残念ながらそういう状態でないため地道にスキャニングしPDF化して保存しておくということになります。

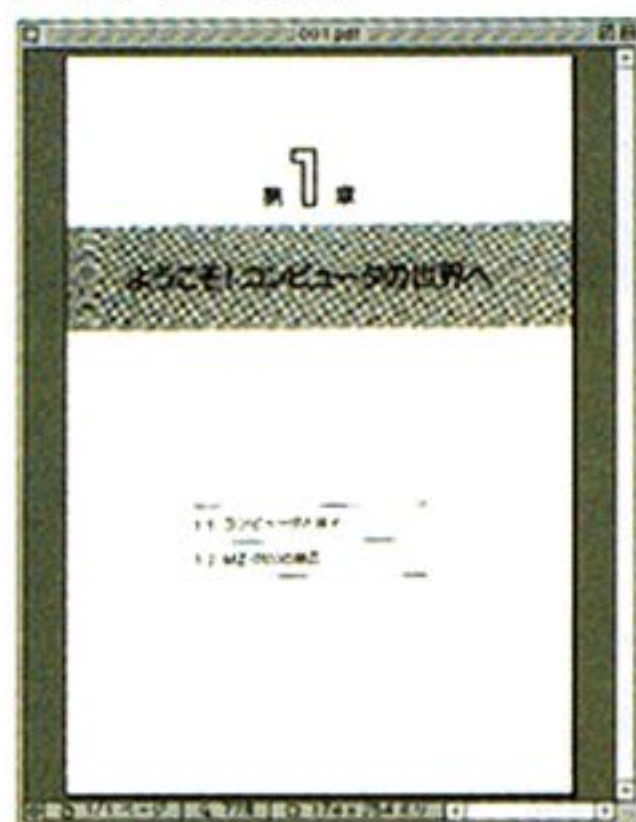
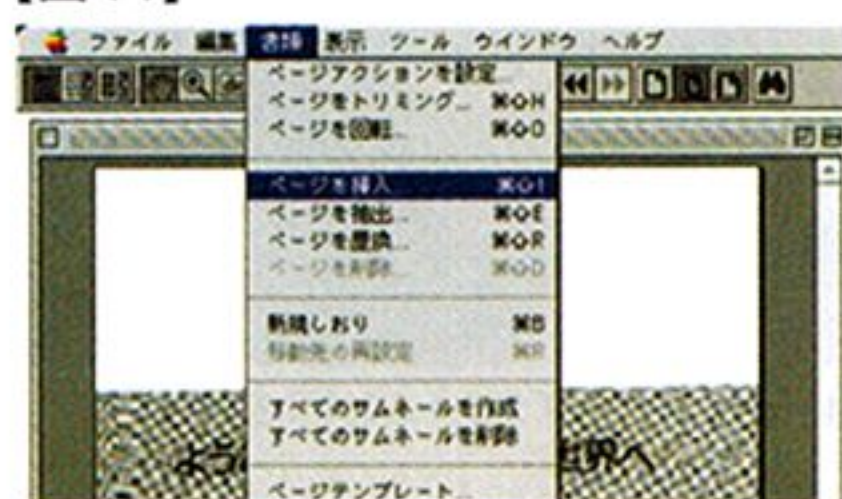
ちなみに実働時間ですがMZ-2861オーナーズマニュアル合計193ページを10時間で処理できました。MZ-700 DISK BASIC(120ページ)のほうは多少早くて8時間程度でした。おおよそ1ページあたり3分で処理できるといったところです。手持ちのマニュアルをPDF化しようと思っている人は参考にしてください。

たとえ過去のものでも，未来への礎となるときがあるかもしれません。すべては積み上げられてきたのですから。

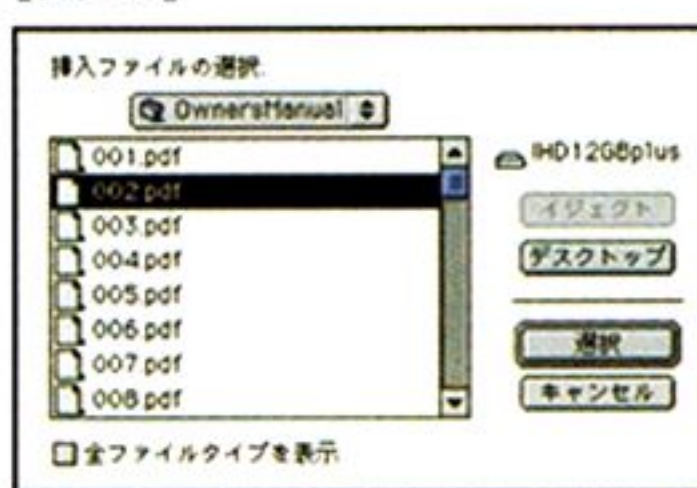
不要マニュアル募集中

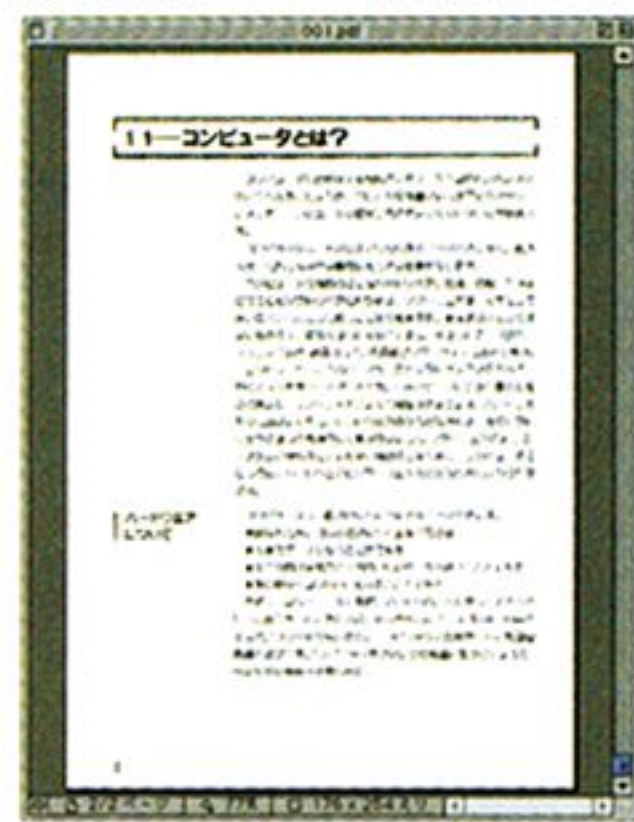
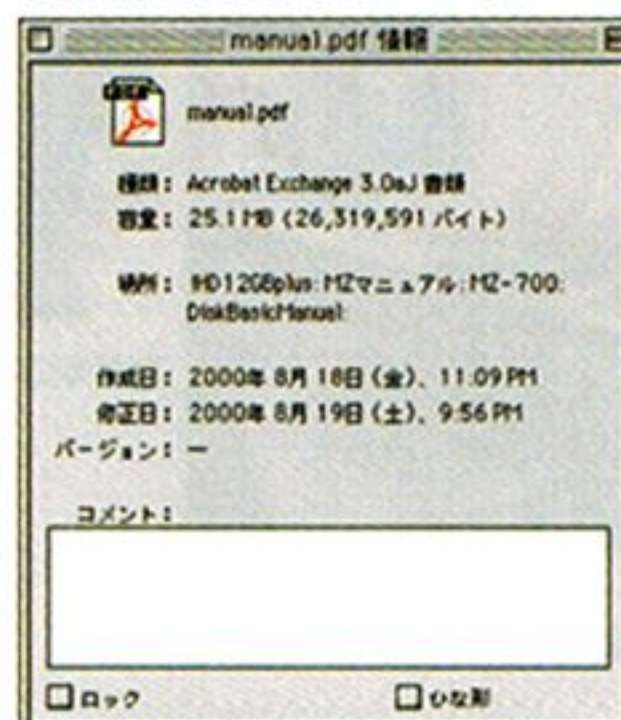
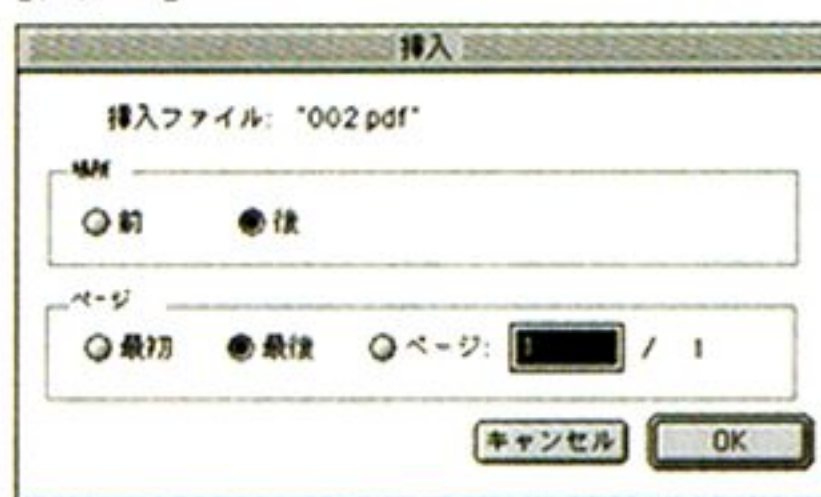
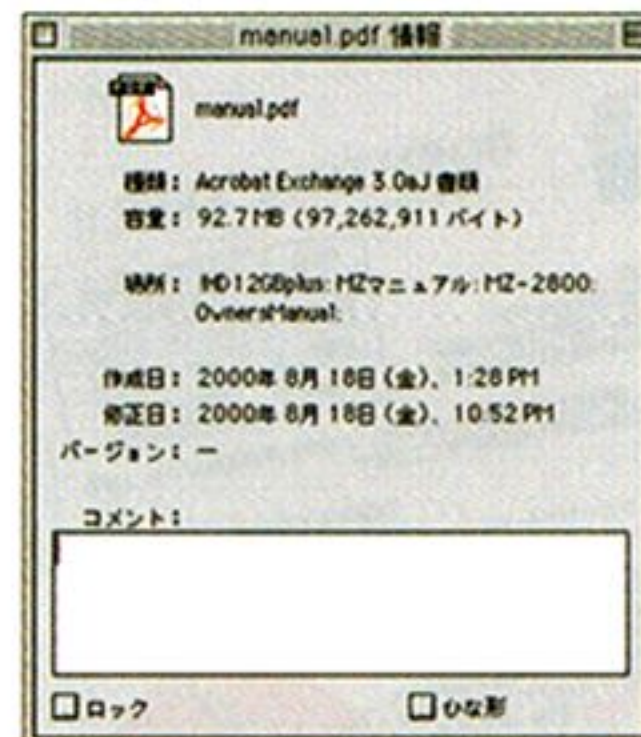
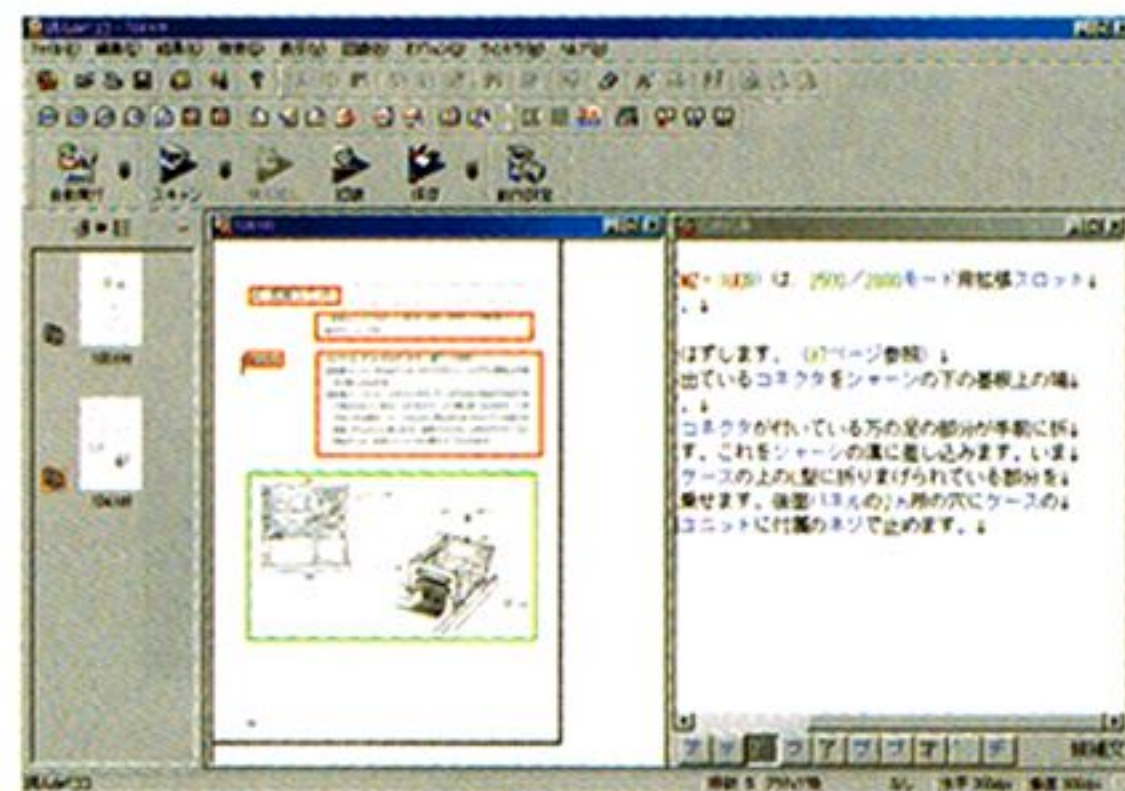
もしMZのマニュアルがあり不要だ，という方は下記メールアドレスまでご連絡ください。引き取ります。

openspc@po.shiojiri.ne.jp
古籏一浩まで。

【図13】
最初のページを開く

【図14】
書類メニューからページを挿入を選択し挿入するページを選択する

【図15】

【図16】

【図17】
OKボタンを押してページの最後に追加する。ページが挿入されると自動的に挿入されたページが表示される

【図18】
MZ-2861オーナーズマニュアル

【図19】
MZ-700 DISK BASICマニュアル

【図20】
取り込んだ画像を文字認識させる。その後，修正を行い保存する。認識率は書体によって異なるが，明朝体では99%か，それ以上，丸文字では70%ほど。ただし，登録しさえすれば99%近くまで認識される

【図21】
PDF化されたマニュアル。文字にアンチエイリアスがかかるので綺麗に見える

【図22】
テキストも選択しコピーなどが可能になる

X68000ユーザーの祭典
第4回フェスタ・68レポート

11月4日，秋葉原のマークインタースペースで第4回フェスタ・68が開催された。X68000単一機種をターゲットとした異色のイベントで，第1回に行ったときよりは人込みは減っているものの，まだまだ元気にやっているという印象。

会場ではハード，ソフト，その他，さまざまな展示物が並んでいるが，なかでも来栖川電工のJupiter-EXが目を引く。Jupiterはいいとしても，そのPCIスロットの上にどかーんと載せられたビデオカード，名前は「SERIKA」……うーむ。そのほか，Dreamcastを超えるSH4ボードとLinuxのコンビネーションなど，同社の展示物はなかなかエキサイティングだ。

ソフト関係はゲームを中心に音楽CDや各種ツールなどが展示販売されていた。しかし全体的に見て，あいかわらず動画モノが多いのはなぜなんだろうなあ。動画再生ソフトだけでいくつあったのだろうか。

恒例となった感のあるストリートファイターXVIでのゲーム大会も，決勝近くになると非常に緊迫した接戦が繰り広げられた。膨大なキャラクターのなかから，まったくタイプの違うキャラが出てきても対応していけるのは相当やり込んでいるということだろう。

コントローラ接続部の不調か，リセットも多発していたが，なんとなく大会専用バージョンの起動画面を見せるためワザとやってんじゃないかと思うくらい誌面に出せないような画面が次々と……。

次回開催は2001年5月5日。ゴールデンウィークは秋葉にGoだ。

これがJupiter-EX。68060/50MHz搭載で13万円（予価）

Jupiter-EX上に載っているのがメガスプライトエンジン搭載ビデオカードSERIKA

SH4ボード上で動作するLinuxのデモ

X68000の筐体内にATXマザーを入れるというもの。そのお隣はATとX68000のマウス/キーボード切り換え器を扱ってる

なぜかニキシ管の時計が（非売品）

ダンスゲー。しかし，コントローラがなければ音ゲーと差がないような気もするのだが

ストリートファイターXVIを使ったゲーム大会の様子

来栖川電工で配られたチラシ

オリゲーフェスタin秋葉原&フェスタ・68のご案内

column

皆さまごぶさたしております。
「フェスタ・68」準備会代表しゃかんきょりたもつでございます。次回の催し物として，
今年5月4/5日にそれぞれ「オリゲーフェスタin秋葉原」「フェスタ・68」を開催いたします！

オリゲーフェスタ in 秋葉原

今回新たな試みとして開催する「オリゲーフェスタin秋葉原」について紹介します。世の中の移り変わりにより，ある機種で実力を磨いていたユーザーはハードにこだわらず，その培った技術を活かすためにほかのマシンへと進出しています。その世の中の流れを汲み取り，ハードの垣根を外してオリジナルゲームならなんでもOKというイベントを開催することになりました。ただこれだけだと，ほかのイベントと同じ内容になってしまいます。そこで出展内容はゲームソフトに限らず，オリジナルハードや開発環境tool，開発に役立つ資料なども募集することで差別化を図りつつ，真のエンジニアが育つ場を作りたいと考えております。

また今後の動向を見極めてオリジナルゲームの意義を拡大し，ロボコンなどソフトだけで楽しむゲームからハードとソフトの両方を駆使して楽しむ遊びを広く世間に流行らせたいと考えております。ぜひ，この実験的ともいえるイベントにご参加くださる，サークルさん&企業さんの参加を心よりお待ちしております！

なお，参加費は完全後払い制ですので，開催当日の9時～10時の間に受け付けまでお支払いください。また当日やむにやまれない事情により参加できない場合でもキャンセル料は一切かかりません。1スペースには，180cmの長机半分の45cm×90cmとチケット2枚をご用意いたします。

以上のような新たなイベントを，準備中ですがこのイベントが無事に成功したときは，「継続的に開催し，いつかは有明に進出できる大きな催し物へと成長させること」が現在の目標となっています。皆様の温かい応援をお待ちしております。

サークル参加もしくは企業参加の募集用に下記アドレスにHPを開設しておりますので，お気軽にご利用ください。

http://y7.net/game/

なお，現在イベント開催当日にお手伝いくださるスタッフも募集しております。詳しくは下記までメールください。

tamotu2@z2.zzz.or.jp

DATA

イベント名称：「オリゲーフェスタin秋葉原」
開催日：2001年5月4日
開催場所：損保会館2階の「大会議室」
（〒101-8335　東京都千代田区淡路町2-9）
一般参加：10時～15時
サークル&企業参加：9時～16時

前売り券は300円，当日券は500円です。前売り券はチケットぴあより，1月15日から販売を開始いたします。

サークルスペース
1スペースのサイズ：180cm×45cmの机半分（3000円）
1サークル当たり最大2スペースまで（6000円）
出展内容：オリジナルゲーム，オリジナルハード，開発環境，開発にまつわる書籍など～

申し込みはHPもしくはFAX（047-368-4288）などから。必要事項は「代表者氏名」「フリガナ」「住所」「電話番号」「サークル名」「フリガナ」「希望スペース数」「HPURL」「リンクの可否（y/n）」「出展内容（できるだけ詳細に）」
＊）企業参加の場合は別途記入ください。

フェスタ・68

「フェスタ・68」も例年どおり開催いたします！ハードの減少とともにユーザー数も減っていますが，一方では新規ユーザーの参入も相次いでおり，とてもよい傾向にあると思います。ただ新規ユーザーの方はDOSを経験したことがない方もいて，少々敷居が高いと思っている人も多いようです。そこで少しでも新規ユーザーのためになるような，基本的な運用方法を皆さんとともに伝えていければと願っています。今回も前回に負けないくらい盛り上げていきましょう！

またイベントカタログ用の原稿も大募集しております。内容は特にX68000に関連しなくても，いまハマっていることやソフト，ハード開発にまつわるお話，または同人ゲームの攻略などなどドシドシご応募ください。応募方法は<tamotu2@z2.zzz.or.jp>に，テキスト形式でお送りください。画像があったり，ファイルサイズが大きい場合は，別途ご相談ください。サークル参加もしくは企業参加の募集用に下記アドレスにHPを開設しておりますので，お気軽にご利用ください。

http://www.pipi.net/X680x0/

なお，現在イベント開催当日にお手伝いくださるスタッフも募集しております。詳しくは下記までメールください。

tamotu2@z2.zzz.or.jp

入場券ですが予算の関係で「オリゲーフェスタin秋葉原」を流用いたします。全国のチケットぴあ取扱店で，「オリゲーフェスタin秋葉原」の券をご購入ください。お値段は通常どおり前売り券は300円，当日券は500円です。前売り券はチケットぴあより1月15日から販売を開始いたします。

DATA

イベント名称：「フェスタ・68」
開催日：2001年5月5日
開催場所：損保会館2階の「大会議室」
（〒101-8335　東京都千代田区淡路町2-9）
一般参加：10時～15時
サークル&企業参加：9時～16時

サークルスペース
1スペースのサイズ：180cm×45cmの机半分（3000円）
1サークル当たり最大2スペースまで（6000円）
出展内容：X68000が関係するソフトやハード，書籍など～

申し込みはHPもしくはFAX（047-368-4288）などで。必要事項は「代表者氏名」「フリガナ」「住所」「電話番号」「サークル名」「フリガナ」「希望スペース数」「HPURL」「リンクの可否（y/n）」「出展内容（できるだけ詳細に）」
＊）企業参加の場合は別途記入ください。

X WindowでX68000のROMフォントを表示する

Mitsuta Tetsuo　三津田 哲雄

2000春号のSTUDIO Xで話に出ていたフォントコンバートに関するものです。MKBDF68kはX68000のCG-ROMフォントから16ドットBDFフォントを作成し，最終的にはXFree86のフォントとして，またMeadowのフォントとして使えるようにしようという目的で作られました。

はじめに

Oh!Xが休刊になってから半年くらい経った頃でしょうか。どうしても時代の流れとプロセッサのスピードには勝てず，いつしかPC-9801で動くPC-UNIX，FreeBSDをメインに使うようになっていました。当然，X68000はあまり使われなくなってしまったのはいうまでもありません。そのFreeBSDには当然UNIX上で動くウィンドウ環境のXFree98(XFree86のPC-98版)を走らせていたのですが，どうも標準で用意されている16ドットフォント，jiskan16が丸っこいのが気になってしょうがありません。というわけでいままで見慣れているX68000のROMフォントをX上で表示させてみようというのがそもそものきっかけです。

というわけで「こんなものでよかったらどうぞ」的な発想で公表することに決定しました。ただ，ほかの方がすでに対応されている可能性は十分にありえますので，あまりツッコミは入れないように。

もちろん，実行ファイル，ソースの類はすべてフリーウェアとして扱っていただいて構いません。

作業に必要なもの

必要なものとして次のようなもの(?)があります。

- X68000本体
(ハードディスクはあったほうがいいでしょう，フロッピーディスクでの運用は少々トリッキーな操作を必要とします)
- rootログインできるUNIXマシン
(もちろんX Windowが動いているマシンです)
- UNIX，X Windowについてある程度の知識
(xfontsel，xfdを使うことができる)
- tar，gzipなどのツール
- X68000魂，ハック魂 :-)

使用環境は FreeBSD+XFree86，Linux+XFree86を想定しています。NetBSD/X68000はX68030が手元にないので厳密にはこうだ！とはいい切れませんが，同じX Window環境ということで，おおむねのところは同じ操作で表示することができるでしょう。適当に読み替えて実行してください。実際，BDFファイル形式でそのまま使えるkonなどでも使えますし，さらにはWindows95/98上で動くXEmacs(Mule)であるMeadowでも設定をすれば使うことができるようになります。

さあ，君のデーモン君やペンギン君を「ぺけろく教」に入信させましょう。

X Windowで使用するフォントについて

PC-UNIX(FreeBSD，Linux)用のフリーで使うことのできるX WindowであるXFree86では，フォントはBDF形式というテキストファイルをPCF形式に変換したファイルをX Window上で使用しています。BDFからPCFへの変換はX Windowの標準パッケージで用意されている(はず)のbdftopcfというコマンドを使って変換します。ただ，これではフォントファイルがディスクのスペースを多く占有してしまうので，実際にはgzipやcompressなどで圧縮してある場合がほとんどです。

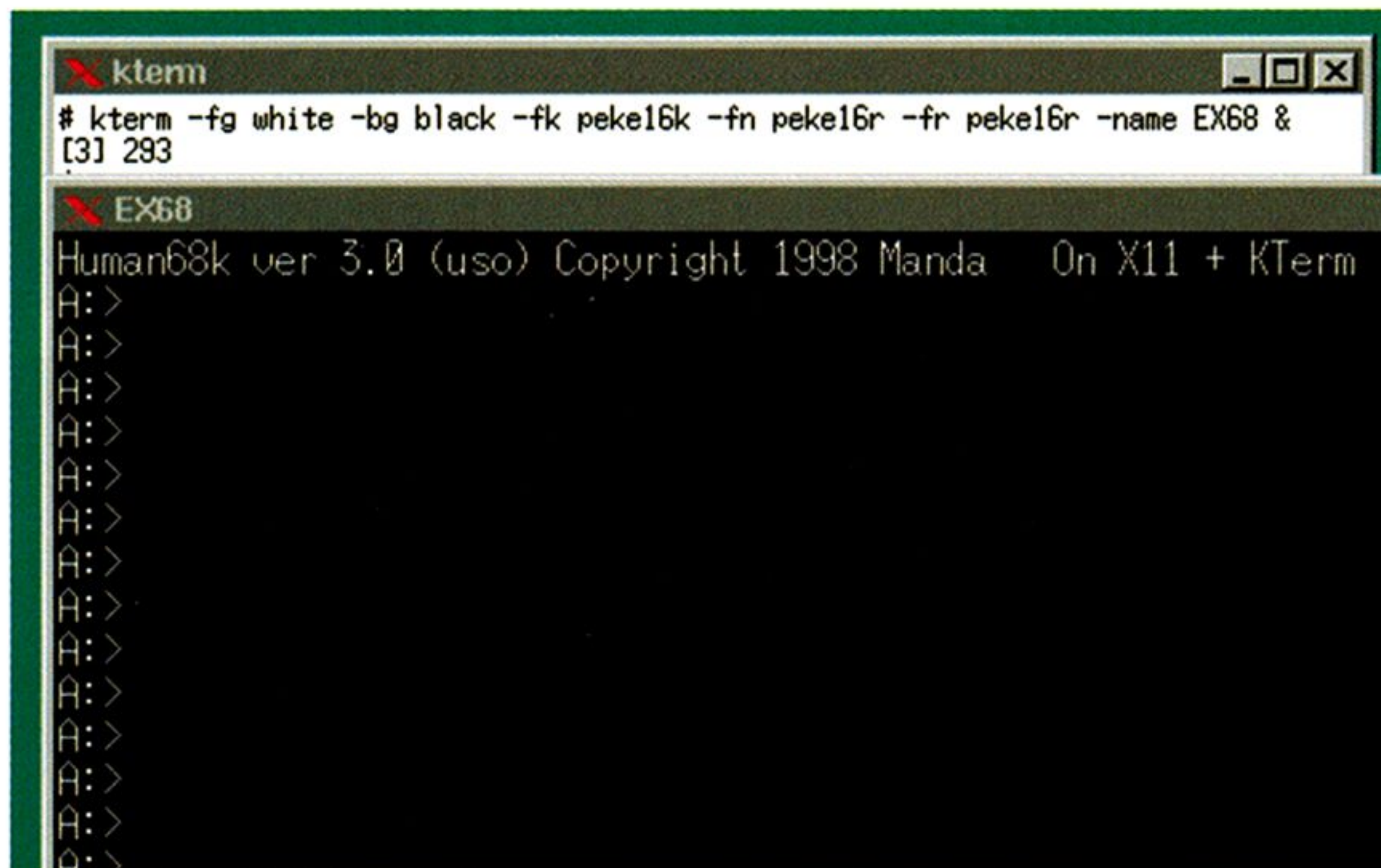
```
% bdftopcf < fontfile.bdf | gzip >
fontfile.pcf.gz
% bdftopcf < fontfile.bdf | compress >
fontfile.pcf.Z
```

もしかしたらほかの環境ではBDFからほかの形式に変換する必要があるかもしれませんが，すべての場合においてBDF形式のファイルさえあれば別の形式に変換できるという点において共通性があるといえます。

ということは，BDFファイルが用意できればX Windowの動作する環境でありさえすれば，好きなフォントを使用できるということで，Mac(MkLinux，NetBSD/m68k)でさえもX68000のROMフォントをX Window上で表示できるということになります(多分)。

実はX Window環境でなくても，DOS/Vマシン用に用意されているkon(漢字コンソール)などの利用によってこのフォントを表示させることもできます。

また，日本語を表示するためには，半角文字のBDFファイル，全角文字のBDFファイルがそれぞれ必要になりますので，これからその2つのファイルをX68000で作成します。

```
kterm
# kterm -fg white -bg black -fk peke16k -fn peke16r -fr peke16r -name EX68 &
[3] 293
EX68
Human68k ver 3.0 (uso) Copyright 1998 Manda   On X11 + KTerm
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
A:>
```

X WindowsのコンソールをX68000風にしてみた

X68000で実際に行う作業

実際に作業するといっても16ドットのBDFフォントファイルを生成するプログラムを実行するだけです。しかしながら，生成されるBDFフォントファイルが大きくなることとディスクアクセスの時間から，ハードディスクは必須になります。もちろん，ROMフォントを吸い出すのでX68000も必要であるのはいうまでもありません。抜け道としてはEX68を動かすという邪道な手もありうるのですが，EX68上の動作確認はとっていませんので注意してください。

とりあえず注意することは，"~"(チルダ)と"¥"(バックスラッシュ，円マーク)の扱いでしょう。SWITCH.Xを使って適切に設定しておきましょう。

実際に使うには，ハードディスクに1.4MB以上の空き領域があることを確認してから，

```
A:>MKBDF16K > PEKE16K.BDF
A:>MKBDF16R > PEKE16R.BDF
```

とすればBDFフォントが生成されます。

今回BDFファイルをテキストファイルとみなしてリダイレクトを使っています。実はこのためにフォントのサイズがPEKE16K.BDFでは1.27MBの大きさになり，2HC(1.2MB)はおろか，2HD(1.23MB)にも入らないこととなります。「困った，フロッピーディスク環境の人は指をくわえてるしかないのか？」そんなことはありません。トリッキーな操作は必要ですが，2つの方法で問題を解決できます。

まずはUNIXでsedによる加工を前提としたBDFファイルを作成する方法です。プログラム実行時に"-t"(Tiny，Tmp)オプションをつけることで，中間ファイルを作るモードになります。違いは改行の代わりに":"を出力しているだけだったりします。

X68000

```
A:>MKBDF16K -t > PEKE16K_.BDF
```

[フロッピーディスクで転送……]

UNIX

```
% sedy/:/"\n"/peke16k_.bdf>peke16k.
bdf
```

次になんとかフロッピーディスクに収まるようにBDFファイルを分割して作成しておき，UNIX側でファイルを結合させるという方法です。これも"-a"，"-b"の各オプションを利用することで2HDフロッピーディスクを2枚使っての作業ではありますが，利用が可能です。

X68000

```
A:>MKBDF16K -a > PEKE16K.BDA
A:>MKBDF16K -b > PEKE16K.BDB
```

[フロッピーディスクで転送……]

UNIX

```
% cat peke16k.bda peke16k.pdb >peke
16k.bdf
```

これらどちらかの方法で転送したファイルで，これからの作業を同じように行っていくことができます。2つ方法を用意したのは一応理由があり，少しでも高速に処理をしたい場合には初めの方法を，sedが見つからないなどの場合に後ろの方法をと思ったからです。もちろん，sedのないUNIX環境はないでしょうが(それではそのマシンは欠陥マシンと呼ばれてもしかたがない)，書いたスクリプトが思ったとおりに動かない可能性はないわけではありません。その場合は後ろの方法を利用してください。

"-t -a -b"のオプションはMKBDF16K.Xに限って有効です。MKBDF16R.Xではそのような指定をしても無駄ですのであしからず。

X WindowでX68000のフォントを表示するまで

これからは実際にフォントを表示させるUNIXマシン上で行います。もちろんシステム内部の変更ということになるのでこれから先フォントの設定までの項目はroot権限での作業になります。

まずは適当な場所にファイルが転送され(圧縮などがされているのであればこの時点で解凍，展開をしておきます)，目的のBDFファイルの名前が，

```
peke16k.bdf
peke16r.bdf
```

であることをlsコマンドなどを使って確認してください。英字はすべて小文字です。ファイルの転送はどんな形式をとってもかまいません。

この時点でフォントファミリなどの変更をした

リスト1 MK_BDF16の処理内容

```
echo off

echo 16 ドットROMフォントからBDF型式のフォントを作成します。
echo 半角 PEKE16R.BDF 全角 PEKE16K.BDF
pause
echo 半角ファイルを作成しています。
mkbdf16r > PEKE16R.BDF

echo 全角ファイルを作成しています……しばらくお待ちください。

if "%1" == "-t" goto tiny
if "%1" == "-T" goto tiny
if "%1" == "/t" goto tiny
if "%1" == "/T" goto tiny

goto full
:tiny
echo 中間ファイルをPEKE16K_.BDFとして作成します。
echo UNIXマシン側でsedストリームエディタを使った操作を必要とします。

mkbdf16k -t > PEKE16K_.BDF

:full

mkbdf16k  > PEKE16K.BDF

echo フォントの作成が終了しました。
```

フロッピーディスクを使ったデータの転送について

column

X68000のフロッピーディスクが3.5インチならばMS-DOS，1.2MB(2HC)フォーマットのメディアを使うことでDOS/V機にも転送ができます。もちろん，1.23MBフォーマットのメディアだと，PC-98x1に転送することができます。PC-UNIXに限っていえば，カーネルから意図的にMS-DOSファイルシステムのサポートを削除しない限り，mountコマンドでMS-DOSフォーマットのフロッピーをマウントできるはずです。5インチFDの場合は適当にメディアコンバートして転送してもらうしかありません。1.2MB(2HC)フォーマットで転送していっても最近のWindowsが動くようなマシンは5"ベイはあっても5"FDDは積んでないので物理的に無理でしょう。

MS-DOSでは8+3文字の制限があり，そのファイル名は大文字でなければならないという点に注意しておきましょう。TwentyOne.xを使ってマルチピリオド，21文字対応，大/小文字の区別ありという環境を作っている方は特に気をつけましょう。

Human68kフォーマットされたディスクはMS-DOSではアクセスが可能ですが，UNIX上ではまずマウントされないでしょう。というわけで，転送の際にフロッピーディスクを使う場合にはMS-DOSでフォーマットしたものを使いましょう。

ほかにもディスクイメージ(MS-DOSのフォーマットではなく)での転送というのもあるらしいのですが，やったことがないのでよくわかりません。

個人的な意見ですが，FreeBSD2.xではVFATのサポートがないものがあります。この問題は3.x あたりではクリアされているようですが，基本的にDOSの8+3文字をもとに命名しています。このほうがわずらわしくないと思いますがどうでしょう。

い方はテキストエディタで変更してください。X Windowではフォントにエイリアスをつけることが可能ですが，たとえばjiskan16.bdfとpeke16k.bdfをそっくりそのまま置き換えたい場合などにはここで編集することになります。

bdftopcfコマンドとgzipまたはcompressを使って変換，圧縮します。

Linux

```
% bdftopcf < peke16k.bdf | gzip >
peke16k.pcf.gz
% bdftopcf < peke16r.bdf | gzip >
peke16r.pcf.gz
```

FreeBSD

```
% bdftopcf < peke16k.bdf | compress >
peke16k.pcf.Z
% bdftopcf < peke16r.bdf | compress >
peke16r.pcf.Z
```

次に，できあがったファイルをX windowの日本語のフォントなどが入っている場所にコピーします。X11R6であれば/usr/X11R6/lib/X11/fonts/misc/です。lsコマンドを使ってファイル名の最後の拡張子が合致しているかどうかを確認してください。

これではまだフォントファイルが登録されないので，mkfontdirコマンドを使ってfonts.dirファイルを更新します。

```
% mkfontdir/usr/X11R6/lib/X11/fonts/
misc/
```

/usr/X11R6/lib/X11/fonts/misc/以外に専用のディレクトリを作ったという場合には，/etc/XF86Configを変更したうえでX Windowを再起動する必要があります。

また，この2つのファイルに対応したフォントのエイリアスを設定します。/usr/X11R6/lib/X11/fonts/misc/fonts.aliasに次の2行を追加します(もちろんこの名前でないといけないというわけではないんですが)。

```
peke16k -peke-fixed-medium-r-normal--
16-150-75-75-c-160-jisx0208.1983-0
peke16r -peke-fixed-medium-r-normal--
16-150-75-75-c-80-jisx0201.1976-0
```

現在作業をX Window上で行っていて，X Windowを再起動させることなく文字を表示するには次のコマンドを実行します。

```
% xset fp rehash
```

これでフォントが表示できるようになりました。xfontsel，xfdを実行して目的のフォントが表示できることを確認します。

```
% xfd -fn peke16r
```

さらなる確認のために，日本語ターミナルエミュレータを起動します(もしあれば)。

```
% kterm -km euc -fk peke16k -frpeke
16r -fnpeke16r &
```

これで日本語のテキストを表示させるなどして正常に表示されれば終了です。X Window上で表示されるX68000のROMフォントを思う存分堪能してください。

BDFファイルから太字のBDFファイルを作成する

そういえば強調文字(ボールド体)のBDFフォントファイルもこの際だから作ってしまいましょう。これから作成の手順をお教えしましょう。これで作成されるファイルが必須かというとそうでもありません。あくまでおまけ的なものとして考えてください。

用意するものはbdfbold.cと先ほど作成したpeke16k.bdf，peke16r.bdfです。特にpeke16k.bdfは中間ファイルではないことに注意してください。すでにこの3つのファイルはUNIXマシン上の適当な場所にコピーされているものとします。

これからの操作はUNIXマシン上で行います。X68000上でもBDFBOLD.Xで作成することも不可能ではないのですが，プロセッサのスピードはUNIXのほうが速いと踏んでUNIXマシン上で処理します。X68000ではハードディスクに最低4MBの空きがないと苦しいでしょう。

まずはボールド体作成用プログラムを作りましょう。

```
% gcc -o bdfbold bdfbold.c
```

エラーなくコンパイルが終了したら，ボールド体のフォントをpeke16rb.bdf，peke16kb.bdfとしてフォントファイルを作成します。コンパイラがパーサエラーを出してしまった場合には，UNIXマシン上のエディタで"^Z(CTRL+z)"をソースファイル上から削除して，コンパイルしてみてください(一応このためにUNIX上でコンパイルできるファイルをtar+gzipで固めたものを収録してあります)。

作成したbdfboldとそれぞれのフォントファイルがカレントディレクトリにあるものとして，次のように入力，実行します。

```
% ./bdfbold < peke16r.bdf|seds/Medium
/Bold/>peke16rb.bdf
% ./bdfbold < peke16k.bdf|seds/Medium
/Bold/>peke16kb.bdf
```

これを先ほどと同じような方法でX Windowのフォントとして登録します。

ちなみにエイリアスは次の通りに設定しましょう。

```
peke16kb -peke-fixed-bold-r-normal--
16-150-75-75-c-160-jisx0208.1983-0
peke16rb -peke-fixed-bold-r-normal--
16-150-75-75-c-80-jisx0201.1976-0
```

sed s/Medium/Bold/はいったいなにをしているのか？　フォントの先頭データ部に次のような加工をしています。察しのいい人なら「こんなのエディタで修正しても同じじゃん」って思っているでしょうね。

MS-DOS / Human68k における改行コードについて

column

MS-DOS/Human68kの改行コードは"CR+LF(0x0A+0x0D)"ということは周知のこととして説明していきます。

MS-DOS/Human68kではファイルモードに主にテキストファイルを扱うTEXTモードとバイナリファイルを扱うBINARYモードがあります。これら2つのOSではファイルを作成する場合には明示しない限りはテキストモードで作成されます。この場合，ファイルに"CR(0x0A)"のみを書き込んだ場合には"CR+LF(0x0A+0x0D)"が自動的に書き込まれるという仕組みになっています。

対して，"BINARYモード"でファイルを作成すると，"CR(0x0A)"を書き込んだとき，そのままの形，"CR(0x0A)"で書き込まれることになります。

これらのことを踏まえたうえで，UN*Xの改行コードは"CR(0x0A)"であることを考えると，少なくともMS-DOS/Human68kではTEXTモードを使うわけにはいきません。たとえばUNIXから持ってきたテキストファイルをTYPEコマンドで出力したものだという場合，余計な"LF(0x0D)"が標準出力に出力されているのです。stdoutは通常TEXTモードでオープンされるのが普通ですから，たとえば"type fugaunix.txt > fugados.txt"とした場合の2つのファイルが違うものになってしまうということがありうるのです。

添付されているcut0d.cは改行コードを"CR+LF(0x0A+0x0D)"から"CR(0x0A)"に簡易変換するプログラムです。標準出力に出力する代わりにそのままファイルに書き込んでいます。

```
FONT -PEKE-Fixed-Medium-R-Normal--16-
150-75-75-C-160-JISX0208.1983-0
WEIGHT_NAME "Medium"

FONT  -PEKE-Fixed-Bold-R-Normal--16-
150-75-75-C-160-JISX0208.1983-0
WEIGHT_NAME "Bold"
```

実は，bdfboldはほかのフォントも加工が可能だという噂もちらほら。かなり厳しい条件がありますが，ソースさえ解読できればなんとかなるでしょう。

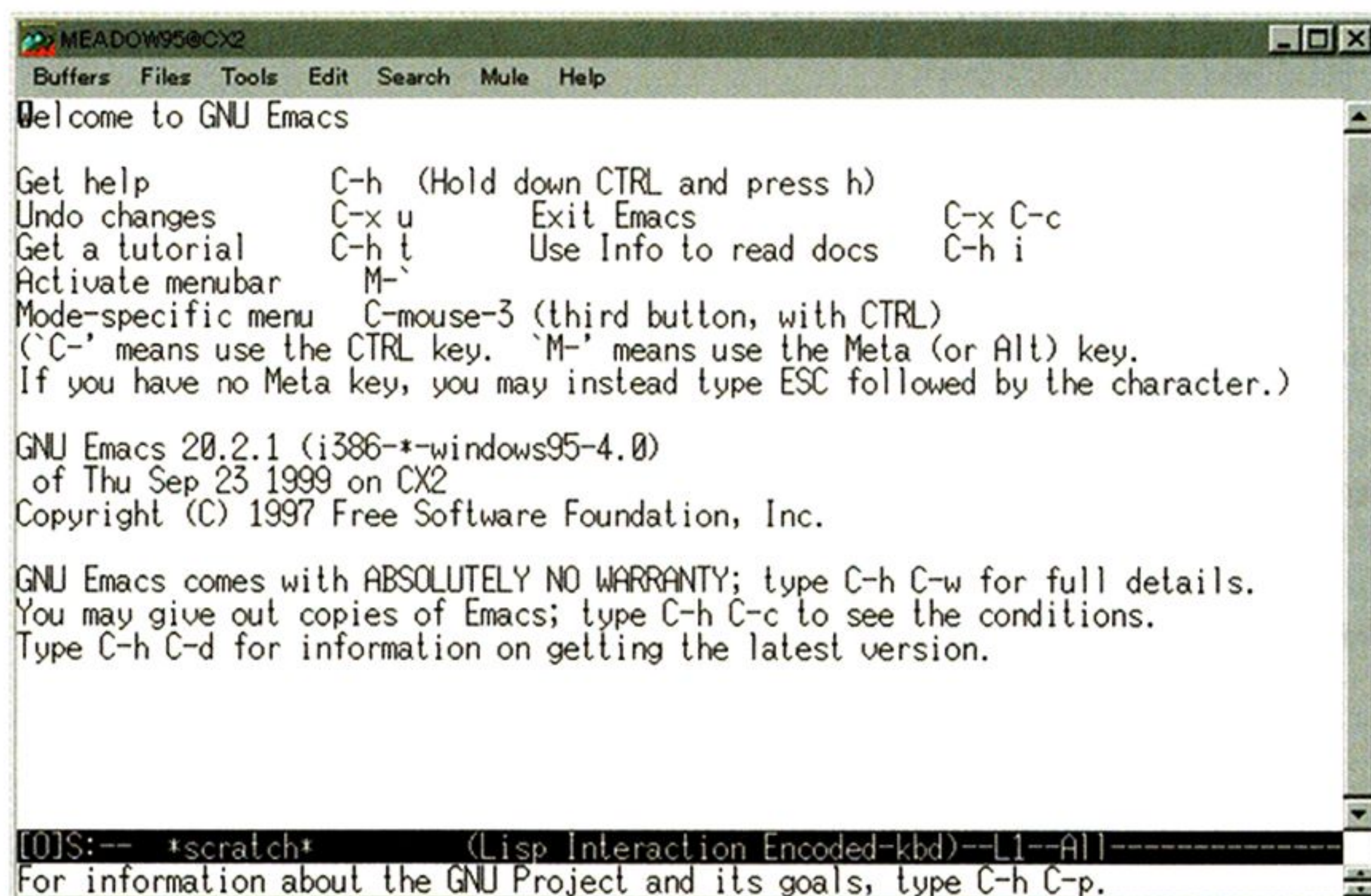
konでX68000のフォントを！

kon（漢字コンソール）でも作成したBDFファイルは使えます。640×480ドットの画面なら80×30のX68000そっくりなテキスト画面ができあがります。

なんらかの理由でX Windowを使えない環境にあり，DOS/V機を使っている人には朗報（?）です。うまく表示を誤魔化せば（プロンプトを変えるのが手っ取り早いでしょう），「ノートで動くHuman68k（もどき）」も作成できます（が，konとノートパソコンの相性はあまりよくないようです）。ここではすでにkonがインストールされた状態であるというのを前提にして話を進めます。

まずは作成したBDFフォントファイルを/usr/share/fonts/peke16/にコピーします。もちろんディレクトリは各自で勝手に「掘って」ください。

```
% mkdir -p /usr/share/fonts/
% cp peke16k.bdf/usr/share/fonts
% cp peke16r.bd/usr/share/fonts
```

kon.cfgを編集します。FreeBSDのパッケージなら/usr/local/etc/kon.confを，RHL（Red Hat Linux）なら/etc/kon.cfgが相当します。

リスト2

```
~/.emacs （抜粋）
  (setup-japanese-environment)
  (set-language-environment "Japanese")

  (mw32-ime-initialize)
  (setq default-inputmethod "MW32-IME")

  (add-hook 'mw32-ime-on-hook
         (lambda  ()   (set-cursor-color "brown")))
  (add-hook 'mw32-ime-off-hook
         (lambda  ()   (set-cursor-color "black")))

  (setq mw32-ime-mode-line-state-indicator "[--]")
  (setq mw32-ime-mode-line-state-indicator-list ' ("--" "[あ]" "[--]" ))

   ;for PEKE16[kr]
    (w32-auto-regist-bdf-font
    "bdf16-japanese-jisx0208-peke" "c:/USR/LOCAL/SHARE/FONTS/BDF/PEKE/peke16k.bdf" 0 )
    (w32-auto-regist-bdf-font
    "bdf16-latin-jisx0201-peke" "c:/USR/LOCAL/SHARE/FONTS/BDF/PEKE/peke16r.bdf" 0 )
    (w32-auto-regist-bdf-font
    "bdf16-katakana-jisx0201-peke" "c:/USR/LOCAL/SHARE/FONTS/BDF/PEKE/peke16r.bdf" 1 )

   ;``peke16'' Font-set  (X68000 CG-ROM FONT)
    (new-fontset "peke16"
       ' ((ascii . "bdf16-latin-jisx0201-peke")
          (japanese-jisx0208 . "bdf16-japanese-jisx0208-peke")
          (latin-jisx0201 . "bdf16-latin-jisx0201-peke")
          (katakana-jisx0201 . "bdf16-katakana-jisx0201-peke")
        ))

    (setq initial-frame-alist
      ' ((foreground-color . "black")
      (background-color . "white")
      (font . "peke16")
   ))
```

```
bdf:Stratup
/usr/local/bin/fld -t bdf n/usr/share
/fonts/peke16k.bdf
/usr/local/bin/fld -t bdf n/usr/share
/fonts/peke16r.bdf
```

次に環境変数を変えてプロンプトを"A:>"に変更します。csh系ならset prompt="A:>"とすればプロンプトは変更されます。sh系ならPS1="A:>"としましょう。もちろんkonを実行しないと画面は変わりません。当たり前です。

Win95/98上のMeadowで使ってみる

Windows95/98で動くEmacs（Mule），Meadow-1.00をX68000のフォントで。このようなことが現実のものとなりました。Mule on X68000のような気分が味わえること請けあいです。実は，この機構，ただ単にBDFフォントをロードしているだけなんですが。

Windows環境ならばEX68でのエミュレーションも捨てがたいものがありますが，たとえばMeadow + Cygwin 環境を使っているWindowsユーザーで「X68000のフォントを使いたい」と思われている方のためのものです。特にPentium/200MHzより遅いプロセッサを使っている方には朗報かと思いますが，いかがでしょうか。もちろん，X68000の画面がそのまま現れるわけではないですけど。

まずはMeadowのインストールから，といいたいところですが，Meadowをインストールするということ自体がかなりDeepな世界のことなんで，Meadowのインストール自体は扱いません。Meadowをインストールしてひととおりの設定，ホームディレクトリの設定などが済んだものとして，.emacsファイルを編集してX68000のフォントを表示してみることに関してのみお話ししましょう。

作成したBDFフォントファイル，peke16k.bdf，peke16r.bdfを，

```
C:\USR\SHARE\FONTS\BDF\PEKE\
```

にコピーします。ここでは便宜上このディレクトリにしていますが，この限りではありません。フォントファイルのありかさえ正確に把握しているのなら以降の設定を適当に読み替えて進めてください。というわけで，

```
半角 BDFファイル － C:\USR\SHARE\FONTS\BDF\PEKE\peke16r.bdf
全角 BDFファイル － C:\USR\SHARE\FONTS\BDF\PEKE\peke16k.bdf
```

がターゲットとなるフォントファイルです。

リスト2のように.emacsファイルが適切に設定されたら，あらためてMeadowを再起動させるとX68000のROMフォントで表示されて

```
MEADOW95@CX2
Buffers Files Tools Edit Search Mule Help
Welcome to GNU Emacs

Get help          C-h  (Hold down CTRL and press h)
Undo changes      C-x u      Exit Emacs            C-x C-c
Get a tutorial    C-h t      Use Info to read docs  C-h i
Activate menubar    M-`
Mode-specific menu   C-mouse-3 (third button, with CTRL)
(`C-' means use the CTRL key.  `M-' means use the Meta (or Alt) key.
If you have no Meta key, you may instead type ESC followed by the character.)

GNU Emacs 20.2.1 (i386-*-windows95-4.0)
 of Thu Sep 23 1999 on CX2
Copyright (C) 1997 Free Software Foundation, Inc.

GNU Emacs comes with ABSOLUTELY NO WARRANTY; type C-h C-w for full details.
You may give out copies of Emacs; type C-h C-c to see the conditions.
Type C-h C-d for information on getting the latest version.

[0]S:--  *scratch*      (Lisp Interaction Encoded-kbd)--L1--All-----------
For information about the GNU Project and its goals, type C-h C-p.
```

Windows用のMeadowのフォントを変更したところ

いるはずです。表示されない場合やおかしな表示になるときには，.emacsファイルをよく確認してみてください。

ソースファイルからのコンパイル

以下のファイルを使用します。

MKBDF16K.C
全角文字(JIS-208)を生成するプログラムのソース
MKBDF16K.ORG
全角文字(JIS-208)を生成するプログラムのソース(オリジナル版)
MKBDF16R.C
半角文字(JIS-201)を生成するプログラムのソース
BDFBOLD.C
ボールド体を作成するフィルタプログラム
CUT0D.C
改行コードを0x0Aにするフィルタプログラム(qkcやnkf，ackの使用できる環境であればまず必要ないものです)

ソースファイルのコンパイルはお決まりのパターン，XC2.0以上の環境で，

```
A:>cc /y MKBDF16K.C
A:>cc /y MKBDF16R.C

A:>cc BDFBOLD.C
A:>cc CUT0D.C
```

としてコンパイルしてください。

フォントデータの吸い出し部にIOCS関数を使っているだけなので，GCCでのコンパイルもライブラリの指定さえきちんとすればコンパイル可能です。'-liocs'の指定を忘れずに。

UNIXでも利用可能なプログラムもあります。

```
% cc -o bdfbold bdfbold.c
% cc -o cut0d cut0d.c
```

として実行してください。

あとがき

ROMフォントはやはりシャープの著作物ということで，シャープの許可をもらわないと吸い出したフォントは配布しないほうがいいですよね。ですから今回はプログラムのみでフォントファイルはありません。すなわち実機が必要だということです。この部分は「実機が手元にある＝X68000のROMフォントに対して愛着がある」と勝手に判断しています(EX68でのエミュレーションは理論上可能なのですが未確認です。HDイメージからどうやってWindows上のファイルに落とすんでしょうか？)。

外字はJIS-212に準処しているかどうかわからないので，いまのところ保留にしてあります。外字フォントも登録するとハドソンの蜂マークなんかも表示できるようになることでしょう。もちろん自前で作成しましょうね。

24ドットフォントについては，まず，よほど大きい画面で使っている人しか使わないだろうということと，差し替えてもそれほど違いのない字体ということで16ドットフォントしか作りませんでした。圧縮してフロッピーディスクに収まるかどうかも怪しいですし(16ドットフォントの圧縮率からして入らないことはないでしょう)。

どうしてファイルサイズが大きくなることがわかっていながら，リダイレクトを使うような，すなわち標準入出力を使用するプログラムを組むかということについては，

・なるべく汎用性を持たせるように

いうことからです。事実，mkbdf16k.cなどはPC-98x1でもROMフォントデータ吸い出し部をPC-98x1専用のものに変更すること＋αで対応が可能なはずですし。その場合のROMフォントはPC-98x1のものですけどね(intの問題をクリアすれば，の話)。さらに，

・なるべく無造作にファイルをオープンして操作しないように気を使った

ことからでしょうか。fprintf関数の使用を嫌った部分もあります。

・この頃，FreeBSD上での作業が多くなっている

こともあり，今回の例ではsed(ストリームエディタ)まで使っていますし。そういう環境に慣れてきているのでしょうね。

・かなり意地の部分もあった

かもしれません。"GNU is Not Unix!!"と叫びながらプログラムを組んだとか組まないとか。

その割にソースが汚いという人は誰ですか！(動きゃいいのよ，動きゃってヤツです。どうせ一度しか起動されないんですから)。

あとがきのあとがき(X11 & UNIX X68000化計画)

現在，twmの配色をグレー系に統一して「Ko-windowみたい」な環境を作っています。あくまで「みたい」な感じの画面に仕上がっています。これに対して少し配色を変えて，vsみたいな感じの画面に仕上げることも可能です。確かWWW上のどこかでSX-WindowやKo-windowに似せたウィンドウマネージャを見たような気もします。それがWindows95用であったかX Window用であったかの記憶が定かではありませんが，そういうものがあればさらにX68000化できるでしょう。アイコンの類はpixmapなどのパターンエディタでしこしこと描いてください。

また，fvwm95やqvwmなどのウィンドウマネージャを使ってX Window上であたかもWindowsとEX68(Human68kの画面だけだけど)が動いているようなスクリーンも作りあげることができます。あとはNexTで動いているように見せかけたり，「MacでもEX68が動くぞ，スクリーンショットもほらこのとおり」っていう悪質なデマが流せたり(ってホントにやらないでくださいね)ということが実現可能です。これには800×600ドット以上の画面が必要ですけど。

```
% kterm -geometry 96x32 -km euc -fk
peke16k -fn peke16r -fr peke16r \
-fg white -bg black -name EX68 &
```

それではさいなら。

使用(テスト)環境：
X68000 EXPERT2 HD (CZ-613C-BK),
Human68k ver 3.0
NEC PC-9801 RA21+IBM486SX3, FreeBSD(98) 2.2.2-R /XFree86 3.2 (XF98_TGUI)
COMPAQ CONTURA400C , Slackware 3.5 + PJE-0.15cm
XFree86 3.3.3.1 (XF86_SVGA), kon2
HITACHI FLORA3010CT , FreeBSD 2.2.8
XFree86 3.3.3.1 (XF86_SVGA), kon2
NEC PC-9821 Cx2 , Windows95/NEC / Meadow 1.0
XC ver2.1 -newkit-
・一応すべてのソースについてX68000上においてGCCでもコンパイル可能です。
・一部のソースはFreeBSD上のgccでもコンパイル，動作確認をしています。

参考資料：

XC ver2.1 -newkit-の各マニュアル
X68000 EXPERT 取扱説明書（シャープ）
XFree86 3.2 document
XFree86 online manual
kon2のドキュメント，Cソース
jiskan16.bdf
UNIX USER（スフトバンクパブリッシング）
ここまでできるFreeBSDパワーガイド（秀和システム）

初めて読むMIPS

第3回　エピソードⅠ

中森 章　Nakamori Akira

この連載は前回で終わる予定だった。しかし，第1回と第2回の記事を読み返してみて気づいたことがある。私自身が日常的にあまりにもMIPSアーキテクチャに親しみすぎていた弊害というか，記述に不親切な部分が見受けられる。68000やx86のいわゆるCISCマイコンの経験しかない人のために，もう少し詳しくRISCの特徴を解説しておくべきだったと反省している。そこで，今回またページをもらって，解説し忘れたMIPSアーキテクチャの特徴を補足しておきたい。

r1と擬似命令

プログラムが利用できる32本の汎用レジスタのうち，いくつかは専用レジスタとしての特別な意味を持っている。r0はゼロレジスタで，値が常にゼロとなるレジスタである。r31はリンクレジスタで，サブルーチンコール時の戻りアドレスが格納される。これらについてはすでに説明した。あと，r29（スタックポインタ），r28（グローバルポインタ）というものがあるが，これらはプログラムに関する決めごとであって，C言語とかのリンクを考えない限りは自由に使用してよい。

このようなレジスタの仲間であるr1は少し特別な意味を持っている。r1の別名はat（assembler temporary）で，アセンブラのマクロ命令で一時的に使用されるレジスタである。これを頭に入れておかないと予期せずにr1の値が破壊されてしまうことがある。実際，アセンブラでr1を使用するとエラーまたは警告が出る。では，プログラムでr1を使用してはいけないかというとそうでもない。通常は使用禁止であるr1もアセンブラの擬似命令で使用可能になる。

```
.set noat
```

というのがそれ（たいていのアセンブラはこのシンタックスである）で，この1行を挿入することで，その行以降でr1が使用可能になる。というか，エラーや警告が出なくなる。プログラマが自覚してr1を使用する分にはなんの問題もない。しかし，誤ってr1を使用することを避けるために，逆の意味の擬似命令もある。それが，

```
.set at
```

である。

NOPと擬似命令

MIPSのアセンブラでは不要なNOP命令の挿入によって性能が低下するのを避けるためか，デフォルトではNOP命令の使用を禁止している。

それでは分岐の遅延スロットに配置するべき命令がない場合や，ロード遅延（R3000の場合）を調節するためにはどうすればいいか。実はなにもしなくてよい。MIPSのアセンブラは命令コードの順序が最適になるように命令を並べ替える機能を持っている。そのときに自動的にNOP命令が挿入されるので，プログラムでは遅延スロットやロード遅延を意識する必要はない。逆に，分岐命令の次になにか命令を置いても遅延スロットとはみなされないので，意図しない結果になる。しかし，命令を勝手に並べ替えてもらいたくない場合や，NOP命令によってタイミング調整を行いたい場合がある。このような場合は擬似命令によって並べ替えを禁止することができる。それが，

```
.set noreorder
```

である。この擬似命令を指定した以後の行からは命令の並べ替えが行われない。NOP命令を使用することもできる。ただし，分岐の遅延スロットやロード遅延は自分で管理しなければならない。逆に，命令の並べ替えやNOP命令の挿入をアセンブラ任せにするには，

```
.set reorder
```

を指定する。この連載では，特に説明してないが，

```
.set noat
.set noreorder
```

が指定されているものとしてサンプルプログラムを紹介している。

乗除算命令

MIPSではほとんどすべての命令を1クロックで処理することを目標としている。当然例外もある。この範疇に当てはまる命令は，浮動小数点演算と一部のシステム制御命令を除けば，乗除算命令がそれだ。乗除算命令は，一般には，1クロックで処理できない。これを通常のパイプラインに組み込むとパイプラインが乱れて性能低下につながる。

これを回避するため，MIPSでは乗除算を通常のパイプラインとは切り離し，ほかの演算と並列に処理するようになっている。このため，乗除算の出力（デスティネーションオペランド）として，汎用レジスタとは別の専用レジスタを用意している。こうすることで汎用レジスタへの依存性をなくする。その専用レジスタがHIレジスタとLOレジスタである。32ビット×32ビットの乗算では積は64ビットであり，上位32ビットがHIレジスタに，下位32ビットがLOレジスタに格納される。同様に，64ビット×64ビットの乗算では積は128ビットであり，上位64ビットがHIレジスタに，下位64ビットがLOレジスタに格納される。

一方，32ビット÷32ビットの除算では32ビットの商がLOレジスタに，32ビットの剰余がHIレジスタに格納される。64ビット÷64ビットの除算では64ビットの商がLOレジスタに，64ビットの剰余がHIレジスタに格納される。

実際のプログラムでは，乗除算命令のあと，数命令後に（乗除算の計算が

図1　乗除算命令の実行クロック数

命令	R3000	R4000	R5000	R10000	R4300	R4100	R4700
mult	12	10	5	6	5	1	6-9
multu	12	10	5	7	5	1	6-9
div	35	69	36	35	37	35	42
divu	35	69	36	35	37	35	42
dmult	N/A	22	9	10	8	4	7-10
dmultu	N/A	22	9	11	8	4	7-10
ddiv	N/A	135	68	67	69	67	74
ddivu	N/A	135	68	67	69	67	74

終了するのを待って)，HIレジスタまたはLOレジスタから結果を汎用レジスタに転送することになる。こうすると，パイプライン処理に乱れは生じない。

HIレジスタの値を汎用レジスタに転送する命令が，

```
mfhi  (Move From HI)
```

であり，LOレジスタの値を汎用レジスタに転送する命令が，

```
mflo  (Move From LO)
```

である。いま，32ビット乗算に3クロック必要と仮定する。この場合は，

```
mult  r2,r3
nop
nop
mflo  r4    // r4にはr2とr3の積(下位)が転送される
```

というように，少なくとも3命令後にHI/LOレジスタを参照することが推奨される。nopの部分は乗除算に無関係な命令の範囲を入れてよい。パイプラインを乱さないために，HI/LOレジスタへのアクセスは乗除算終了後に行うことが推奨されてはいるが，これは強制ではない。プログラム的には乗除算命令の直後からmflo/mfhiを置くことも可能である。その場合は，パイプラインはインタロックし，乗除算の実行終了まで待ち合わせが生じる。MIPSアーキテクチャではパイプラインをインタロックしないことが特徴である(特にR2000/R3000では)が，この場合は唯一の例外である。図1に各MIPS　RISCにおける乗除算処理の実行クロックを示す。

MIPSアーキテクチャでは汎用レジスタからHI/LOレジスタに値を転送する命令もある。それが，

```
mtlo  (Move To LO)
mthi  (Move To HI)
```

である。プログラム上はこのような命令は不要である。しかし，HI/LOレジスタは，汎用レジスタと同じく，タスクを特定するコンテキストの一部なので，タスク切り替えの際にOSが使用する。

乗除算命令はデコード時にソースオペランドのリードと同時に実行が開始される。パイプライン的に乗除算命令の前2命令の範囲にはmflo/mfhi/mtlo/mthiを置くことはできない。mflo/mfhiは乗除算の途中結果を参照する可能性があり，mtlo/mthiは乗除算の途中結果を破壊する可能性があるためである。もっとも，(特にアウトオブオーダーな)スーパースカラを採用すると2命令前という制限が無意味になるので，R10000などではこのような制限はない。

ところで，乗算の結果を特殊レジスタに格納するMIPSの方式は使いにくいのか，MIPS32/MIPS64アーキテクチャではデスティネーションに汎用レジスタを指定できる乗算命令であるMULが導入された。MUL命令は3つのオペランドを持ち，そのひとつがデスティネーションレジスタとなる。たとえば，

```
mul  r3,r1,r2   // r3 <- r1 × r2
```

のように記述される。ただ，不可解なのはMUL命令は32ビット×32ビットの乗算のみのサポートで，64ビットの乗算はMIPS64の範囲でも存在しない。MIPSが決めた命令にしてはエレガントさを欠いている(DMULがあってもいいはず)。噂ではMIPS32を初めて実装するJadeを開発したのはLSI Logic社の技術者であるという。事実，LSI Logic社のTinyRISCシリーズのIPコアではMIPS32の拡張命令であるMADD/MADDU/MSUB/MSUBU/MUL命令を早い段階で実装していた(CLZ/CLO命令は未実装)。そこら辺に理由があるのではないだろうか。

サブルーチンコール

CISCでいうところのサブルーチンコール命令はスタックに戻りアドレスを退避してターゲットであるサブルーチンや関数にジャンプする。しかし，暗黙のうちにスタックを使用するCISC式のサブルーチンコール命令は，ロード/ストア命令を基本とするRISCとは相容れない。そこでRISCでは戻りアドレスをスタックではなく汎用レジスタに格納してターゲットにジャンプするのが一般的である。MIPSアーキテクチャでもこの方式を採用する。

```
jal  (Jump And Link)
```

という命令がそれで，戻りアドレス(遅延スロットがあるのでJALのアドレス＋8番地)を汎用レジスタであるr31に格納してターゲットにジャンプする。JAL命令では戻りアドレス(リンクアドレスという)を格納する汎用レジスタはr31に固定されているが，ターゲットアドレスを汎用レジスタで指定する，

```
jalr  (Jump And Link Register)
```

命令ではr31以外のレジスタを指定することも可能である。しかし，一般にはr31が使用される。

さて，r31はひとつしかないのでサブルーチン内でさらにサブルーチンコールを行う場合はr31を退避しておかなければならない。通常，退避場所にはスタックが用いられる。たとえば，次のような命令シーケンスになる。

```
    jal    Target1
    :
    :

Target1:
    addiu  r29,r29,-4
    sw     r31,0(r29)      // r31をスタックに退避
    :
    :
    jal    Target2         // 別のサブルーチンをコール
    :
    :
    lw     r31,0(r29)      // r31をスタックから回復
    nop                    // ロード遅延
    jr     r31             // リターン
    addiu  r29,r29,4       // 遅延スロット
```

なお，ロード遅延回避のためのnop命令はR4000以降のMPUでは，自動的にインタロックするので不要である。インタロックしない場合，ロード命令では，パイプライン的に，2命令後でないとロードした値を使用できない。これがロード遅延である。一般的には，スタックにはローカル変数用のレジスタも退避されるので，サブルーチンからのリターン時に，

```
    lw     r31,0(r29)      // r31を回復
    lw     r25,4(r29)      // ローカル変数用のレジスタを回復
    jr     r31
    addiu  r29,r29,8
```

などと，早めにr31を回復してロード遅延を回避している。

jal命令はサブルーチンに無条件に分岐する。MIPSアーキテクチャでは条件サブルーチンコールというべき命令も存在する。つまり，条件(レジスタの値が0以上か，0より小か)が成立する場合にのみ分岐する命令である。jal命令と同様に戻りアドレスをr31に格納する。これらについてはあとで条件分岐と同時に説明する。

条件分岐命令

CISCでは条件分岐命令といえば，ゼロ，符号，キャリなどの条件フラグを参照して分岐の成立/不成立を判断する。しかし，MIPSアーキテクチャでは，通常は，条件フラグというものを定義せず2つの汎用レジスタの値を比較してその大小関係で分岐の成立/不成立を判断する。これは条件分岐命令の種類を削減するためではないかと推測される。たとえば，条件フラグが

4ビットあれば16通りの組み合わせが考えられ，その組み合わせの数だけ条件分岐命令が必要である。レジスタの比較だと，=，≠，≦，≧，<，>の6種の関係だけでこと足りる。まあ，条件フラグを定義すると，汎用レジスタと同様に，パイプラインのステージ間でフォワーディング（値のバイパス）が必要になり，制御回路が複雑になるのを避ける意味もあるのかもしれない。

1）レジスタ比較による分岐

MIPSアーキテクチャで定義されている，レジスタ比較による条件分岐は，

```
beq     (Branch on EQual)
bne     (Branch on Not Equal)
blez    (Branch on Less than or Equal to Zero)
bgtz    (Branch on Greater Than Zero)
bltz    (Branch on Less Than Zero)
bgez    (Branch on Greater or Equal to Zero)
bltzal  (Branch on Less Than Zero And Link)
bgezal  (Branch on Greater or Equal to Zero And Link)
```

の8種である（正確にはLikely分岐があるので16種）。簡単な英語なので意味はわかるだろう。最後の2つはサブルーチンコール用の条件分岐である。beq/bne以外は単一の汎用レジスタとr0（ゼロレジスタ）との比較である。2つの汎用レジスタ間の大小比較で条件分岐する場合は条件セット命令と組み合わせて使用する。条件セット命令は，

```
slt     (Set on Less Than)
sltu    (Set on Less Than Unsigned)
```

の2種類であるが，組み合わせる条件分岐命令をbeq/bneで使い分けることで逆の条件もテストできる。つまり，

```
slt   r2,r3,r4       // r3<r4ならr2に1を格納
bne   r2,r0,target // r3<r4ならtargetに分岐
```

の逆の条件は，

```
slt   r2,r3,r4       // r3<r4ならr2に1を格納
beq   r2,r0,target // r3≧r4ならtargetに分岐
```

でテストできる。なお，条件セット命令にはイミディエート値との比較である，

```
slti    (Set on Less Than Immediate)
sltiu   (Set on Less Than Immediate Unsigned)
```

も用意されている。

一方，MIPS16モードの条件分岐は0との比較である，

```
beqz    (Branch on Equal to Zero)
bnez    (Branch on Not Equal to Zero)
bteqz   (Branch on T is Equal to Zero)
btnez   (Branch on T is Not Equal to Zero)
```

の4種しかない。Tというのはr24のことで，r24の値が0か否かで条件分岐する。MIPS16において2つの汎用レジスタ間の関係で条件分岐するには，減算命令か排他的論理和命令と組み合わせて使用する。MIPS16では比較命令も用意されており，これは2つの汎用レジスタの排他的論理和をr24に格納する。

ただし，この場合，汎用レジスタの値の一致/不一致しかテストできない。値の大小をテストするには条件セット命令を使用する。条件セット命令は比較結果（0か1か）をr24に格納する。MIPS16では2オペランド命令が基本なのでr24という条件格納用のレジスタを使用する。これは，一種の条件フラグといえなくもない。

2）条件フラグによる分岐

MIPSアーキテクチャにおいてコプロセッサの条件判定には条件フラグを使用する。コプロセッサの制御レジスタ内に最低1ビットの条件フラグ（条件コードという）を持ち，その値が0か1かで分岐の成立/不成立を決める。そのための条件分岐命令が，

```
bczt   (Branch on Coprocessor Z is True)
bczf   (Branch on Coprocessor Z is False)
```

である（Likely分岐も存在する）。ここで，zは0または1である。アーキテクチャ的にはzの値として2も定義されるが，その挙動はインプリメント依存である。zが0，つまりコプロセッサ0（CP0）はMPUに内蔵されているシステム制御ユニット，zが1，つまりコプロセッサ1（CP1）はFPU（浮動小数点演算ユニット）である。

CP0の場合，条件コードはステータスレジスタのビット18（CHビット）である。このビットはR4000/R4400以外では特別な意味はない。ソフトウェアでリード/ライト可能である。使い道はユーザー（というかOS）任せである。R4000/R4400では最後に実行した2次キャッシュに対するキャッシュ命令がキャッシュのタグにヒットしたか否かを示す。使い道はよくわからない。

CP1の場合，条件コードはCP1内のステータスレジスタ（FCR31）にある。通常は1ビット（ビット23）であるがMIPS IVでは8ビットに拡張された。これらのビットは浮動小数点比較命令の比較結果が格納される。たとえば，

```
c.eq.s   fp0,fp2   // fp0とfp1の値が等しいとき条件コードが1
nop
bc1t     target    // 条件コードが1なら分岐
```

のように使用する。なお，古いプロセッサでは，bc1t/bc1fでの条件コードのサンプリングは直前の命令の実行中に行われるので，比較命令と条件分岐命令の間には少なくとも1命令を挿入しなければならない。

CP0ハザード

パイプライン動作を行っているMPUは，見かけ上は各命令が1クロックで処理されているように見えるが，命令自身のレイテンシ（命令をデコードしてからレジスタにライトバックするまでの時間）は5クロック程度である。通常は直前の命令の結果を次の命令で使用することはできない。それをRISCではフォワーディングなどの技術で可能にしている。しかし，プログラムの実行で頻度の低い命令や性能に直接関係のない命令では無理をして複雑な待ち合わせ制御（インタロック）をする必要はない。性能に影響しないところではできるだけ手を抜く（といっては言葉が悪いが）のがRISCの信条である。

たとえば，OSでしか使用されないCP0関係の命令の処理がそうである。具体的にはMTC0命令で変更したシステム制御レジスタの値を直後のMFC0命令で参照できるとは限らない。また，MTC0命令でステータスレジスタを変更して割り込み不可に設定したとしても直後の命令から割り込みを受け付けるとは限らない。つまり，ある命令の作用がMPU全体に及ぶには時間がかかるということである。この時間をMIPSアーキテクチャではCP0ハザードと呼ぶ。OSなどシステムの挙動を操作するプログラムではこのCP0ハザードを考慮して，ソフトウェアで明示的に待ち合わせをする必要がある。図2，図3に，それぞれ，R4000，R4300のCP0ハザードの一覧を示す。なお，R3000ではCP0ハザードは一律2～3クロックとして定義されている。

図2，図3において，「ステージ」とは命令を実行するときのパイプラインステージの仮想的な番号である。図ではMPUが特定のソースまたはデスティネーションにアクセスするステージを表してある。たとえば，命令Aの副作用が命令Bに及ぶ場合，CP0ハザード数は，

（命令Bのデスティネーションのステージ）－（命令Aのソースのステージ）

で計算できる。その値から1を引いたものが2命令間に必要なクロック数（NOP命令の数）になる。

たとえばR4300では，同じシステムレジスタを参照するMTC0命令とMFC0命令の間には（7－4）－1＝2個のNOP命令が必要である。もっとも，最近のMPUではCP0ハザードは減少する方向にあり（ハードウェア的にインタロックする），R10000ではCP0ハザードはなくなった。現状では，最新のMPUを使用する場合，CP0ハザードをほとんど考慮する必要がなくなりつつある。プログラマにとって便利な方向にMPUのほうが進化するのが世の中の趨勢か。本当は，アウトオブオーダーなスーパースカラにおいては，命令と命令の間のNOPはほとんど無意味なので，（不本意でも）ハードウェアのほうで面倒を見なければならないというのが実情である。

CP2とCP3

MIPSアーキテクチャではCP0はシステム制御コプロセッサ，CP1は浮動小数点演算ユニット（FPU）であることはすでに述べた。しかし，アーキテクチャではCP2とCP3も定義されている。厳密にいうと，R2000/R3000/

図2　R4000のCP0ハザード

操作	ソース		デスティネーション	
	名前	ステージ	名前	ステージ
MTC0	gpr rt	3	cpr rd	6
MFC0	cpr rd	4	gpr rt	6
TLBR	Index TLB	5-7	PageMask EntryHi EntryLo0 EntryLo1	7
TLBWI TLBWR	Index/Random PageMask EntryHi EntryLo0 EntryLo1	5-7	TLB	7
TLBP	EntryHi EntryLo0 EntryLo1	3-6	Index	6
ERET	EPC/ErrorEPC Status	4	Status Status.EXL Status.ERL	8 4
IndexLoad Tag	—		TagLo TagHi	8
命令フェッチ	EntryHi.ASID Status.KSU Status.RE Config.K0 Config.IB	0	—	
命令フェッチ 例外	—		EPC Status Cause BadVAddr Context	7 2
コプロセッサ テスト	Status.CU	2		
割り込み	Status.IM	3		
ロード ／ストア	EntryHi.ASID Status.KSU Status.RE Config.K0 Config.DB	4	—	
ロード ／ストア 例外	—		EPC Status Cause BadVAddr Context	7

図3　R4300のCP0ハザード

操作	ソース		デスティネーション	
	名前	ステージ	名前	ステージ
MTC0	gpr rt	4	cpr rd	7
MFC0	cpr rd	4	gpr rt	7
TLBR	Index TLB	5-7	PageMask EntryHi EntryLo0 EntryLo1	8
TLBWI TLBWR	Index/Random PageMask EntryHi EntryLo0 EntryLo1	5-8	TLB	8
TLBP	EntryHi EntryLo0 EntryLo1	3-6	Index	7
ERET	EPC/ErrorEPC Status	4	Status Status.EXL Status.ERL	4-8
IndexLoad Tag	—		TagLo TagHi	8
IndexStore Tag	TagLo TagHi	7	—	
命令フェッチ	EntryHi.ASID Status.KSU Status.RE Config.K0 TLB	0 2	—	
命令フェッチ 例外	—		EPC Status Cause BadVAddr Context	8 3
コプロセッサ テスト	Status.CU	2		
割り込み	Status.IM	3		
ロード ／ストア	EntryHi.ASID Status.KSU Status.RE Config.K0 Config.DB	4	—	
ロード ／ストア 例外	—		EPC Status Cause BadVAddr Context	8

R4000ではCP2とCP3が定義されていたが，R5000/R10000（MIPS IV）以降ではCP3の命令コードはCOP1X（拡張FPU）に割り当てられたので，CP2のみが残っている。さて，このCP2とかCP3とはどういうコプロセッサであろうか。実はこれといった規定はない。一応，

```
COP2    (命令コードのビット31-26が010010であるものすべて)
mfc2    (Move From Coprocessor 2)
mtc2    (Move To Coprocessor 2)
dmfc2   (Move Doubleword From Coprocessor 2)
dmtc2   (Move Doubleword To Coprocessor 2)
cfc2    (move Control From Coprocessor 2)
ctc2    (move Control To Coprocessor 2)
bc2t    (Branch on Coprocessor 2 is True)
bc2f    (Branch on Coprocessor 2 is False)
bc2tl   (Branch on Coprocessor 2 is True Likely)
bc2fl   (Branch on Coprocessor 2 is False Likely)
```

という命令コードは定義されているので，これを利用して各メーカーが独自にオリジナルなコプロセッサをインプリメントできる。たとえば，PlayStation2に使用されているEmotionEngineではCP2にベクトルユニットを割り当てている。これ以外にCP2やCP3を積極的に活用している例は知らない。たいていのMPUではCP2やCP3の命令コードを実行しようとしたときの挙動は未定義である。多くの場合，コプロセッサ使用不可例外ではなく，予約済み命令例外が発生するはずである。

さて，R2000/R3000にはCpCondという外部端子が4本あり，それぞれCP0からCP3の状態に対応している。CpCond［0］やCpCond［1］の値はbczt/bczf（z＝0,1）命令でソフトウェアから参照できる（CP1であるFPUは外付けであったことに注意）。特にCpCond［0］はbc0t/bc0f命令と組み合わせて外部回路と同期をとるために使用できる。

それでは，CpCond［2］やCpCond［3］はなにかというと，実はマルチプロセッサストールのために割り当てられていた。マルチプロセッサストールとは，マルチプロセッサ構成が可能なように，MPUの外部からアドレスを指定してデータキャッシュの値をリードしたり無効化するための仕組み（いわゆるバススヌープ）である。具体的には，CpCond［3］はパイプラインを強制的にストールさせるために使用し，CpCond［2］はそのときの操作（リードか無効化か）を指定する。どうやらCpCond［2］やCpCond［3］はコプロセッサとは直接関係ないようである。その端子状態がbc2t/bc2f/bc3t/bc3f命令で参照できるか否かは不明である。

リバースエンディアン

メモリに格納するデータの形式にはバイト並びの違いによって2とおりがある。データの下位バイトをアドレスの小さいほうから格納するのがリトルエンディアン，データの上位バイトをアドレスの小さいほうから格納するのがビッグエンディアンである。

このいい方はコンピュータ用語のうちではそれほど古くない。1980年にDanny Cohenが『On Holy Wars and a Plea for Peace』という論文の中で初めて使用したというのが定説である。その語源はジョナサン・スイフトの『ガリバー旅行記』にある。小人国（リリパット）の中に出てくる，ゆで玉子を小さい端から食べる（割るだったかも）主義の人々と大きい端から食べる主義の人々が由来である。「端」を表す「エンド」という単語に，「主義者」を表す「イアン」（例としてベジタリアンなどがある）が合成されてできた。昔は，リトルエンディアンをインテル形式，ビッグエンディアンをモトローラ形式と呼んでいた。ちょっと前のMIPSのドキュメントではSEX（性別）と書かれていた時期もある。エンディアンという表現は，日本では坂村健氏辺りが広めたような気がする。

エンディアンといえば難しく考える人が多いが，MPUのデータバスとメモリのバイトごとの結線のやり方が異なるだけである。ビッグエンディアン

でもリトルエンディアンでもデータバス上では同じイメージになる（図4）。演算器などはデータの下位側から計算をしていくので，その意味で，すべてはリトルエンディアンに集約されるといえなくもない。まあ，エンディアンとは，あくまでもメモリ上にデータがどの順序で格納されているかを示しているにすぎず，MPUの内部処理とは直接は関係ない。

また，ビッグエンディアンの場合，ビット番号の名づけ方がリトルエンディアンと逆順になっていて惑わされやすいが，実質（バイト内のビットイメージ）は同じである。

現状，x86系のMPUはかたくなにリトルエンディアンを守り続けているが，後発のMPUの多くはリトルエンディアンとビッグエンディアンの両方で動作することが可能である。このような形式をバイエンディアンと呼ぶこともある。MIPSアーキテクチャのMPUは基本的にバイエンディアンである。リセット時にリトルエンディアンで動作するかビッグエンディアンで動作するかを指定することができる。ただし，いったん動き始めたらエンディアンは固定される。

さて，ユーザーが自身のシステムに採用しているMPUを新しいものに切り替えようとするとき，新しいシステムでも古いプログラムで使用していたデータを使いたいと思うときがある。しかし，古いシステムで使用していたOSのサポートするエンディアンと新しいシステムで使用するOSのサポートするエンディアンが異なる場合は，過去のデータをそのまま利用できない。

図4　データバス（32ビット）とメモリの結線

“どちらもデータバス上では0x12345678となる”

(a)リトルエンディアン

MPU	データバス	メモリ	アドレス
MPU	D[31:24]	0x12	4N+3
	D[23:16]	0x34	4N+2
	D[15:8]	0x56	4N+1
	D[7:0]	0x78	4N+0

(b)ビッグエンディアン

MPU	データバス	メモリ	アドレス
MPU	D[31:24]	0x12	4N+0
	D[23:16]	0x34	4N+1
	D[15:8]	0x56	4N+2
	D[7:0]	0x78	4N+3

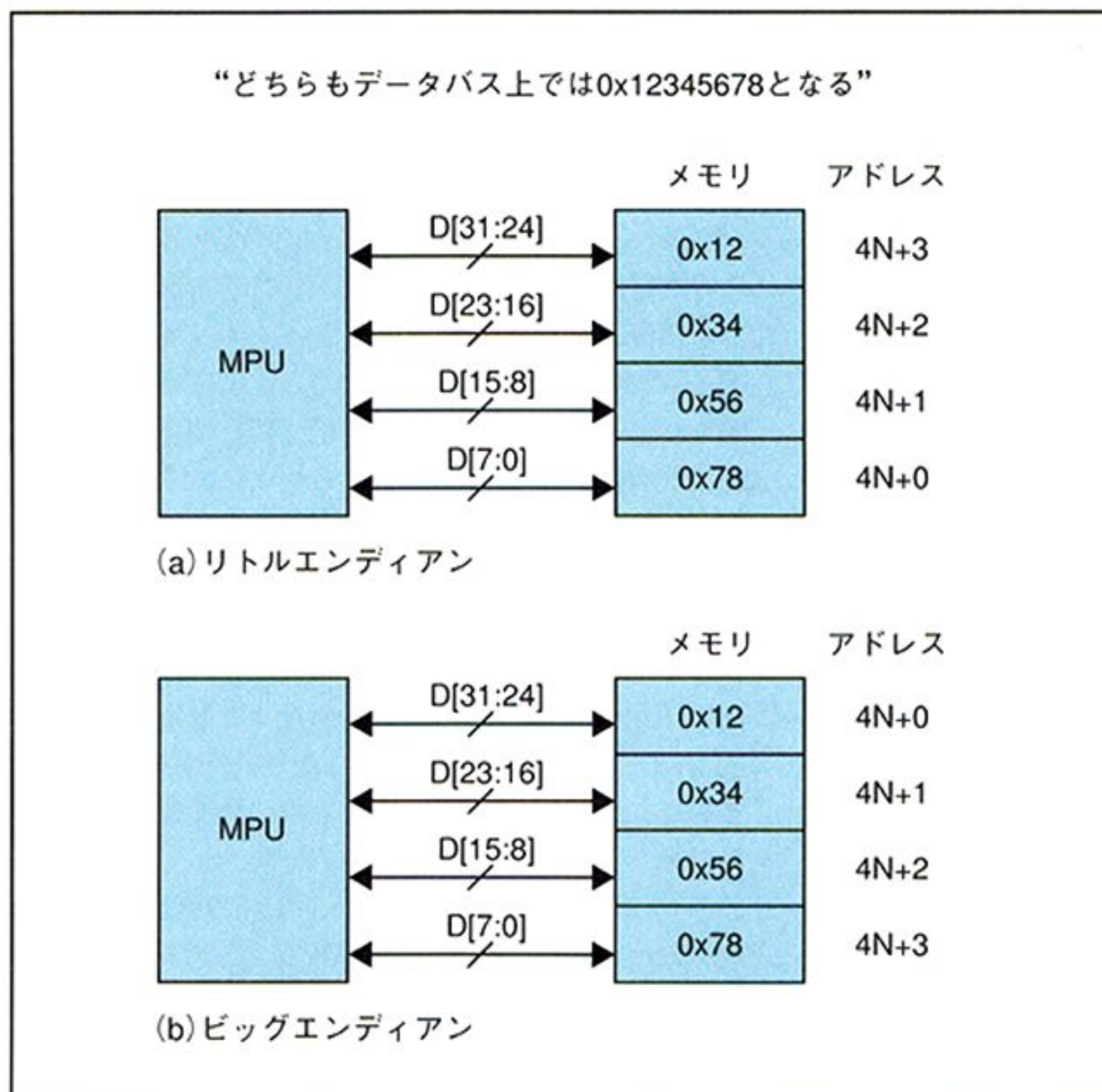



図5　32ビットバスのリバースエンディアンのアドレス

データ長	アドレス	バイトイネーブル	RE時のアドレス
バイト	0	□□□■	3
	1	□□■□	2
	2	□■□□	1
	3	■□□□	0
ハーフワード	0	□□■■	2
	2	■■□□	0
トライバイト	0	□■■■	1
	1	■■■□	0
ワード	0	■■■■	0

エンディアンの異なるシステムへの移行を容易にするため，MIPSアーキテクチャではリバースエンディアンという機能がある。これは，ユーザーモード（アプリケーションプログラム）でエンディアンを反転する機能である。

リバースエンディアンの指定はステータスレジスタのREビット（ビット25）で行う。ステータスレジスタはタスクごとに固有なので，MIPSのシステムではリトルエンディアンのタスクとビッグエンディアンのタスクを混在させてマルチタスクを行うことができる。これはOSがそのようなタスクの混在を許可している場合に限られる。もっともそんなOSはほとんど存在していないのが現状である。その理由として，あとで述べるように，リバースエンディアンの仕様が確定していないこともあると思う。

エンディアンは一度決定したらダイナミックに変更することはできない。正確にいうと，MPUとメモリの結線の方法はシステムに固有なので，エンディアンを途中で変更することは不可能である。アプリケーションプログラムはリトルエンディアンまたはビッグエンディアンのどちらか一方でコンパイルされディスクに保存されている。それが，プログラムの起動時に，システムに定められたエンディアン（バイト順序）に従ってメモリに格納されて実行される。このため，ビッグエンディアンでコンパイルされたプログラムをリトルエンディアンのシステムのメモリに格納すると，メモリ上でのバイト順序が想定しているのとワード単位で逆順になってしまう。

リバースエンディアンとは，この逆順に格納されたプログラムを動作させるために，あたかもエンディアンが反転したかのように「見せかける」技術である。

具体的には出力するアドレスの下位2ビットまたは3ビットを適当に切り替えることで実現する。データバス上の有効なバイト位置（バイトイネーブ

図6　64ビットバスのリバースエンディアンのアドレス

データ長	アドレス	バイトイネーブル	RE時のアドレス
バイト	0	□□□□□□□■	7
	1	□□□□□□■□	6
	2	□□□□□■□□	5
	3	□□□□■□□□	4
	4	□□□■□□□□	3
	5	□□■□□□□□	2
	6	□■□□□□□□	1
	7	■□□□□□□□	0
ハーフワード	0	□□□□□□■■	6
	2	□□□□■■□□	4
	4	□□■■□□□□	2
	6	■■□□□□□□	0
トライバイト	0	□□□□□■■■	5
	1	□□□□■■■□	4
	4	□■■■□□□□	1
	5	■■■□□□□□	0
ワード	0	□□□□■■■■	4
	4	■■■■□□□□	0
5バイト	0	□□□■■■■■	3
	3	■■■■■□□□	0
6バイト	0	□□■■■■■■	2
	2	■■■■■■□□	0
7バイト	0	□■■■■■■■	1
	1	■■■■■■■□	0
ダブルワード	0	■■■■■■■■	0

ル）はエンディアンによって変わらないので，これで十分である。まさに，玉子が語源だけに，コロンブスの玉子的発想である。図5，図6にR3000（32ビットバス），R4000（64ビットバス）でのリトルエンディアンとそのリバースエンディアンのアドレスの切り替わり方を示す。ここではリトルエンディアンを基本にしたがビッグエンディアンでも同様である。

図5，図6に示すアドレスの切り替えでエンディアンの反転はうまくいきそうである。しかし，R4300の開発時にR4000のアーキテクチャを流用しようとしたときに不都合が生じた。R4000からR4300への変更でいちばん大きいのはデータバス幅が64ビットから32ビットになったことである。データバス幅を基準にアドレス切り替えを行うリバースエンディアンでは，データバス幅が違うMPUで互換性を持たせようとすると，命令フェッチで破綻をきたす。64ビットのデータバスで32ビット長の命令のフェッチを行う場合，バイトイネーブルの変化を考えると，リバースエンディアン時のアドレスの下位ビットは，

4→0→12→8→………

の順に変化する。これを32ビットバスで行うと，メモリからの命令フェッチ

活況を呈するMIPSプロセッサ

2000年4月21日，（特許侵害の係争沙汰もあり，MIPS社と仲の悪い）Lexra社は，MIPSアーキテクチャの新しい32ビットIPコアを発表した。LX4189がそれで同社のLX4180コアの後継に当たる。その特徴としては，IPコアとしては初めて250MHz以上の動作周波数を実現したということがある。具体的には，266MHzで動作し，250MIPSを達成する。クリティカルパスを削減するために，パイプラインの先頭に命令メモリアクセスステージを設け，パイプラインのステージ数を従来品の5ステージから6ステージに変更した。CPUコアの面積は1 mm²（0.15 μルール）で結構小さい。発表文を見ると，MIPS社のJade（4Kc）コアを意識していることは明らかで，依然としてMIPS社との間に確執があることを匂わせている。事実，LX4189では件の訴訟の焦点となっているLWL/LWR命令はサポートしていないとのこと。どちらにしても性能はそこそこで高性能という感じはない。お手軽さを狙ったものか。

一方，5月30日，IDT社はRC32334というシングルチップを発表した。MIPS32命令セットを実装し，性能は150MHz動作時に197MIPSという。性能はそんなに高くない。その特徴は最大66MHzで動作するPCIインタフェイスである。ボードでの実装面積の縮小と低価格を狙ったという。これもひとつの方向性と思うが，価格が150MHz品で24ドルというのは，最近のMPUの低価格化の傾向を考えると少し高い。

さらに，6月1日，SCE社はPS2のチップセットであるEmotionEngineとGraphicsSynthesizerを外販すると発表した。これはPS2発表の当時から一部のマスコミで噂されていたことであるが，SCEはノーコメントというかむしろ否定的だったような気がする。いろいろ，もっともらしい説明がされているが，PS2だけでは製造ラインを独自に設計した元が取れないのだろうか。まあ，MIPS32でもMIPS64でもない128ビット命令セットが明らかになる日も近いか。EmotionEngineの動作周波数は300MHz，MIPS/MHzは1.5程度なので，整数性能は取り立てて高性能というわけではないが，6.2GFLOPSという浮動小数点性能は魅力である（でも，単精度のみなのでCGくらいにしか利用できないが）。もっとも，NDA（Non-Disclosure Agreement）を締結しなければ仕様を公開してもらえないなら一般に流布するとは思えないので，一般への情報公開をどの程度行うか，SCEの裁量に期待したい。

6月5日，LSI Logic社は，自社のEasyMACRO ASICコアの新しいラインアップとして，EZ4021 MiniRISCコアとEZ4103 TinyRISCコアを発表した。どちらも，いま流行の論理合成可能なIPコアで，LSI Logic社の0.18 μmプロセスで製造されることを前提としている。EZ4021の特徴はもっとも高速（250MHz）な64ビットコアである。250MHz以上で動作する32ビットコアとしては，Lexra社のLX4189があるので，あえて64ビットと断っていると思われる。16Kバイトの命令キャッシュと16Kバイトのデータキャッシュ込みで12mm²（3.5mm平方）というのは，まあまあ小さい。消費電力は2.6mW/MHzというから，250MHzでは650mWである。組み込み用途を狙っている割には少々大きいか。性能は275MIPS。性能はそれなりであるが，Lexra社のLX4189よりは高い。一方，EZ4103は低消費電力と低価格（≒小面積）を特徴とする32ビットコアである。85MHz品のEZ4102の高速版にあたる（恐らくは単純シュリンク）。命令セットはMIPS IIであるが，プログラム容量削減のためにMIPS16にも対応している。120MHzで動作し60mW（0.5mW/MHz）という低消費電力であり，コアサイズも1.9mm²（1.4mm平方）以下とかなり小さい（キャッシュの面積は含んでいないと思われる）。こちらは結構お買い得かもしれない。ただ，EZ4103の性能はピーク時に120MIPS，平均で96MIPS（120MHz動作時）と少々低いのが気になる。

最後に，6月12日から5日間にわたってサンフランシスコで開催されたEmbedded Processor Forumでも，MIPSアーキテクチャのプロセッサがいくつか発表された。このフォーラムは，1999年にはSCEのEmotionEngineやインテルのStrongARM2が発表されるなど，業界での注目度は高い。ひとつ目はMIPS社の発表によるMIPS64アーキテクチャのR20000で，CPUコアはあのRuby（20Kc）である。MIPS-3D命令セットをサポートし，ゲームや組み込み制御分野を目指している。性能は750MHz動作時に1500MIPS，3.0GFLOPSで，グラフィック性能は37Mポリゴン/秒という。0.18 μmプロセスで製造した場合の動作周波数は600MHz，0.15 μmプロセスなら750MHzという。キャッシュのウェイ予測とマイクロTLBで低消費電力化を実現したという割には500MHz動作時に2Wとそんなに低い電力ではない。コアサイズも0.18 μmプロセスで34mm²（5.8mm平方）とかなり大きい。当初の予定通り昨年末の発表なら少しはインパクトがあったかも。MIPSは世界最高速のIPコアを謳っているようであるが，モノが出てからいいましょう。

2つ目は，DEC社でAlphaとStrongARMを開発していた技術者が設立したSiByte社のSB-1コアである。こちらは，20Kcと同じくMIPS64とMIPS-3Dを実装し，1GHz動作で2.5Wの低消費電力という。まさに，Alpha（高速）とStrongARM（低消費電力）のいいとこ取りである。インオーダーな4ウェイ（メモリ2ウェイ，演算2ウェイ）スーパースカラ構造で，1GHz動作時に2000MIPSという。0.15 μmプロセスで製造した場合のコアサイズは25mm²（5mm平方）と，まあ妥当な大きさである。結構期待していいかも。

3つ目はSiByte社と同様に，StrongARMの開発者が設立したAlchemy社のAu1000である。MIPS32命令セットを実装し，最高500MHzで動作するという。400MHz動作時の消費電力は0.5Wと小さい。単位電力当たりの性能が900MIPS/Wというから，400MHz動作時の性能は450MIPSということになる。低消費電力化のためにスーパースカラ構造，投機実行，分岐予測は採用せず，データキャッシュはウェイ予測を行っている。SiByteにしろ，Alchemyにしろ，ARMのあとに（期せずして？）どちらもMIPSアーキテクチャを採用している点が興味深い。

4つ目はLexra社のNetVortexである。こちらはネットワーク分野に特化したシステムである。従来のネットワークプロセッサとは，標準のMIPSアーキテクチャでプログラム可能なこと，ライセンス可能なソフトコアである点が異なる。NetVortexのCPUはLX8000という名称で，スペース削減のためにMMUやFPUといったルータに不要な機能は実装しない。また，データキャッシュの代わりに，ソフトウェアで管理する2ポートメモリを内蔵している。4スレッドをサポートするLX8000は16KBの命令キャッシュと16KBのデータメモリ込みで，0.18 μmプロセスで製造した場合，3.4mm²（1.8mm平方）という。動作周波数はソフトコアで250MHz，ハードコアで427MHzを目標としている。

このように，次々とMIPSアーキテクチャのCPUコアが発表されるということは，アーキテクチャの完成度の高さ，あるいは設計のしやすさを表しているのかもしれない。ただ，IPコアとしての発表が大半で，製造ラインを持たない会社が論理設計だけ（つまり頭の中で考えただけ）で頑張っている感もする。論理シミュレーションでの性能が実際に達成できるか否かは少し疑問である（って，毎回いっているなあ）。

が逐次的ではなくワード(32ビット)単位で逆転して見える。ところが，32ビットバスで命令フェッチを行う場合，リトルエンディアン時でもビッグエンディアン時でもアドレスの下位ビットは，

0→4→8→12→………

と逐次的に変化するのでリバースエンディアンとの不整合が生じる。したがって，R4300のリバースエンディアンは，命令フェッチが意味的に完全にはエンディアン反転しないので，実用性がない。これは64ビットバスのR4000(正確にはその後継のR4200)と互換性を持たせたために生じた不都合である。このように，一部のMPUにおいて，現在では，リトルエンディアンのリバースがビッグエンディアン，ビッグエンディアンのリバースがリトルエンディアンという(意味上の)関係が不明確になってしまっている。

MIPS16の神話

巷には，MIPS16モードを使用するとシステムの性能が向上するという神話がある。これについて考察しよう。

MIPS16では，命令長が基本的に16ビットになる。このため，命令長が32ビットのネイティブモード(仮にMIPS32と呼ぶ)に対して，命令フェッチにおける単位バスサイクル当たりの転送命令数が倍になる。また，小規模システムでデータバスが16ビット幅の場合に，効率的な命令フェッチが可能になる。といったイメージが浮かび，それに伴い，命令の処理速度も向上すると思ってしまう。しかし，これは誤りである。

命令フェッチは命令キャッシュから1命令単位に行われるので，外部バスのビット幅には影響を受けない。MIPSアーキテクチャのMPUを製造しているLexra社の試算によるとMIPS16コードにすることで命令のコードサイズはMIPS32の60%に縮小されるという。それはそれで嬉しいことであるが，MIPS16にしたからといってデータサイズは変わらない。トータルなサイズとしてどの程度節約できるか疑問がある。

また，命令のコードサイズに関しても50%にならないということは，命令数としては1.2倍(60%÷50%)に増加していることになる。基本的に1命令の実行時間は1クロックというのは変わらないので，単純に計算しても，命令の処理時間は20%増加する。

実際には，イミディエート値やディスプレースメントのビット数を拡張するEXTEND命令は16ビット命令と結合されて32ビット長になるが，EXTENDされた命令の実行には2クロックを要するので，さらに実行時間は長くなる。加えて，プログラムで使用できる汎用レジスタが，MIPS32では32本であったのに対して，MIPS16では8本に減少するので，必然的にレジスタのメモリへの退避/回復の頻度が増加する。これも性能低下の要因になる。

こう考えてくると，MIPS16を積極的に採用する理由は見つからない。あえて利点を探すとしたら，命令セットが比較的Z80に似ているので，歴史的に十分にこなれたコンパイラ技術を適用できるということくらいか。実際に，レジスタをあまり使用しないようなアプリケーションではMIPS16のコンパイラは結構優秀なコードを出力する。といっても，MIPS32のコンパイラ技術はそれよりもさらに優れていると思うのだが。なにしろ，RISC全盛の牽引力となったのがMIPSコンパイラであるという説もあるくらいだから。

MIPS16とよく似た経緯で誕生したARMアーキテクチャのThumb命令セットでは，コードサイズは70%になるが処理時間は45%低下するというのがARM社の見解である。MIPS16にもこれと同じことがいえるだろう。経験的にはMIPS32の半分程度の性能になるときもあるが，ほとんど変わらない場合もある。平均的に45%の性能低下ということはない。せいぜい，20～30%の性能低下か。

ついでに，日立のSHシリーズのMPUで検証しておこう。SH3，SH4，SH5の性能をMIPS/MHz(≒IPC＝Instructions Per Cycle＝1クロックに実行できる命令数)で示すと，それぞれ，

```
SH3   1.3   (Dhrystone2.1?)
SH4   1.8   (Dhrystone1.1)
SH5   1.79  (Dhrystone1.1)
```

となっている。SH4はSH3より38%性能がよい。これがスーパースカラの性能向上分と考えられる。SH5は，SH4がスーパースカラだったのに反し，シングルパイプラインになった。このとき，SH5とSH4のIPCはどちらも1.8程度なので，シングルパイプラインになった性能低下を考慮すると，SH5では実質的に38%の性能向上である。

では，SH5とSH4の違いは何か。前者は命令長が32ビット(汎用レジスタは64本)，後者は命令長が16ビット(汎用レジスタは16本)。つまり，性能の違いが命令長にあると考えることはできないだろうか。SHシリーズでも32ビットの命令長のほうが16ビットの命令長より38%性能がいいのである。結局，命令長を切り詰めるということは，命令コードサイズを削減する意味しかないと思われる。

おわりに

MIPSアーキテクチャの解説もこれで本当に(多分)最後である。ただ，この連載を読み直して項目の構成がスムーズな流れに乗っていないということを反省している。連載を読み直す際には，第1回，第3回(今回)，第2回の後半，第2回の前半の順に読めばスムーズに理解できるのではないだろうか。

なお，この連載ではMIPS RISCのパイプラインについてはあまり言及していない。パイプラインについては，別の連載である『コンピュータアーキテクチャ　その直感的アプローチ』を参照してほしい。その第3回ではシングルパイプライン構造のR3000とR4000を解説している。また第4回ではスーパースカラ構造のR10000のパイプラインを解説している。それでMIPSアーキテクチャの基礎は完璧である(かな？)。バスインタフェイスなどを解説してもよいのだがMIPSアーキテクチャのプロセッサが次々とIPコア化されている現在では，CPUコアの切り口のバスインタフェイスはユーザーに見えてこないので，覚えてもあまり役に立たない。

MIPSアーキテクチャに関してここが知りたいという要望があれば愛読者カードでお願いしたい。いっておくがPS2のEmotionEngineの追加命令に関してはなにも知らないので，ご要望には応えられない。逆にタレコミ情報を待っている。それでは。

どんどん拡張されるMIPSアーキテクチャ

2000年7月25日，MIPS社はスマートカードメーカーのGemplus社と共同で，スマートカードの市場に乗り出す計画を発表した。同時に，MIPS32を拡張し，スマートカードに適した，低消費電力，高性能なSmartMIPSというアーキテクチャを開発するという。

また，SmartMIPSはSUN Microsystems社のJAVAカードにも対応すると発表されている。さらに，MIPS社は，2000年10月9日から開催されるMicroProcessor Forumではネットワーク処理に特化したアーキテクチャ拡張を発表する予定であったが，直前にアジェンダ(論題)から削除された。結局ひっそりと発表はされていたようだ。。同時に，SiByte社のSB-1250というネットワークプロセッサの発表があるので，このCPUに実装されているアーキテクチャと推測される。CPUコアは先に発表されたSB-1なので，こちらはMIPS64を拡張したものなのであろう。

MIPS16，MDMX，MIPS-3Dと自身のアーキテクチャを拡張してきたMIPS社であるが，こうなると，命令数はどんどん増加し，「少ない命令数」というRISCの範疇を完全に逸脱してきている。おそらく，各組み込み制御分野ごとにMIPSアーキテクチャを拡張していくのがMIPS社の新しい戦略なのだろう。そのうち，GameMIPSとか，携帯電話MIPSとかが発表されたりするのかもしれない。

国産アーキテクチャ SH5の野望 ～全世界版～

船本昇竜　*Funamoto Shoryu*

残り少ない国産CPUの筆頭といえるのがSHシリーズだ。ここではDreamcast2で採用か？　とかつて騒がれた64ビットプロセッサ，SH5について概要を紹介してみたい。大きな舞台に立つことはなくなったものの，プロセッサとしてはさまざまな意欲的試みも取り入れられているチップだ。

かつてDreamcast2に搭載されるであろうといわれていたSH5について語ってみたい。単純に家庭用ゲーム機用プロセッサとして見た場合にはEmotionEngineだとかGekkoなどの，ひどく手強いライバルが存在したことは紛れもない事実であったし，Dreamcast2自体が頓挫してしまったことでSH5の位置づけは宙ぶらりんになってしまった。しかし，SH5には純国産マイコンSHシリーズならではの魅力もあり，それにより開拓される市場もあるだろう。

ここでは，そんなSH5の予習をしてみたいと思う。

正真正銘64ビットプロセッサ

SHシリーズは5世代目にして，ついに「正真正銘の64ビットプロセッサ」へのステップを踏んだ。開発している日立がそういっているのだから間違いない。もっとも，詳しくは後述するが，同時処理できる最大ビット数はそのビット数の何倍もある。

64ビットになっても，「SHシリーズらしさ」は変わらない。コード/実行効率を優先した「完全固定長命令」は健在だ。性能・消費電力・集積度・チップサイズのバランスも配慮され，「製品の一部としてのSH」という特徴も相変わらずだ。

●計4096ビットの余裕のレジスタ

さて，その64ビット化の内容であるが，とにかくすぐ足りなくなる汎用レジスタが64ビットになり，数も64本になった。詳しくは後述するが，命令によっては，1本の64ビットレジスタを2本の32ビットレジスタ，4本の16ビットレジスタというように扱うこともできるので，「汎用レジスタ」として，かなり扱いやすくなったといえるだろう。なんといっても，システム固定用途レジスタは何本か存在するものの，SH4のレジスタバンクのように無理やりひねり出したモノとは違うため，なにより感覚的に気持ちがよい。ちなみに，今回は待望（？）の「汎用レジスタ上の零レジスタ（ハードウェアで値が0に固定されているレジスタ）」が加わった。しかし，その配置がR63（最終レジスタ）となっている。慣れないと，少し違和感があるものの，システムレジスタで実現すること（たとえばZレジスタなんてのを新しく用意する）よりマシだろう。

●好きな人も多いからカタログスペック

SH5は日立とSTマイクロエレクトロニクスとの共同開発で，0.15μmプロセス技術を用いて，SH8000シリーズとして発売される予定。紙の上のスペックでは400MHz以上の動作で700～1000MIPS，2.8～4.0GFLOPS，1.6～2.4GMACS，9.6G～14.4GOPSとなる。いろいろな数字が出てきて煙に巻かれそうである。なお，それぞれの単位は，

MIPS(ミップス)：Million Instruction Per Second
GFLOPS(ギガフロップス)：Giga FLoating-point Operation Per Second
GMACS(ギガマックス)：Giga MACexecution per Second
GOPS(ゴップス/ゴープス)：Giga Operation Per Second

である。

ここで少し気になるのが，InstructionとOperationってどう違うのか？ということ。SH5は何種類かのSIMD（シムド：Single-Instruction Multi-Data）命令を持っており，ひとつの命令で複数のデータを処理することができる。詳しくは後述するが，SH5には，ひとつの命令で合計24の算術演算処理を1サイクルで完了する命令がある。400MHz駆動時において，1秒間にこの命令（Instruction）は400M（4億）回しか実行されないが，算術演算（Operation）の回数は9.6G（96億）回にものぼる。

●32ビット×2unit対64ビット×1unit

字面のスペックだけを見ると，SH4時代は32ビット2wayスーパースカラだったのが，64ビットシングルスカラとなり，FPUもオプションとなった。一見すると，いろんな部分で，パフォーマンスが落ちたのでは？　と考えてしまいがちではある。というより「スーパー」という文字がなくなり，ハッタリが利かなくなったというべきか。

図1　SH5 CPUコア

浮動小数点演算装置（浮動小数点レジスタ：64本×32ビット，オプション）
整数/マルチメディアユニット（整数/SIMDレジスタ：64本×64ビット）
命令キャッシュ（32KB，4ウェイセットアソシアティブ）
MMU I-TUB（64エントリ）
データキャッシュ（32KB，4ウェイセットアソシアティブ）
MMU D-TUB（64エントリ）
32
64
64
2×64
64
64
命令フロー/デコードユニット
ロード/ストアユニット
割り込み
SHdebug
32+64
2×64
SuperHywayインタフェイス

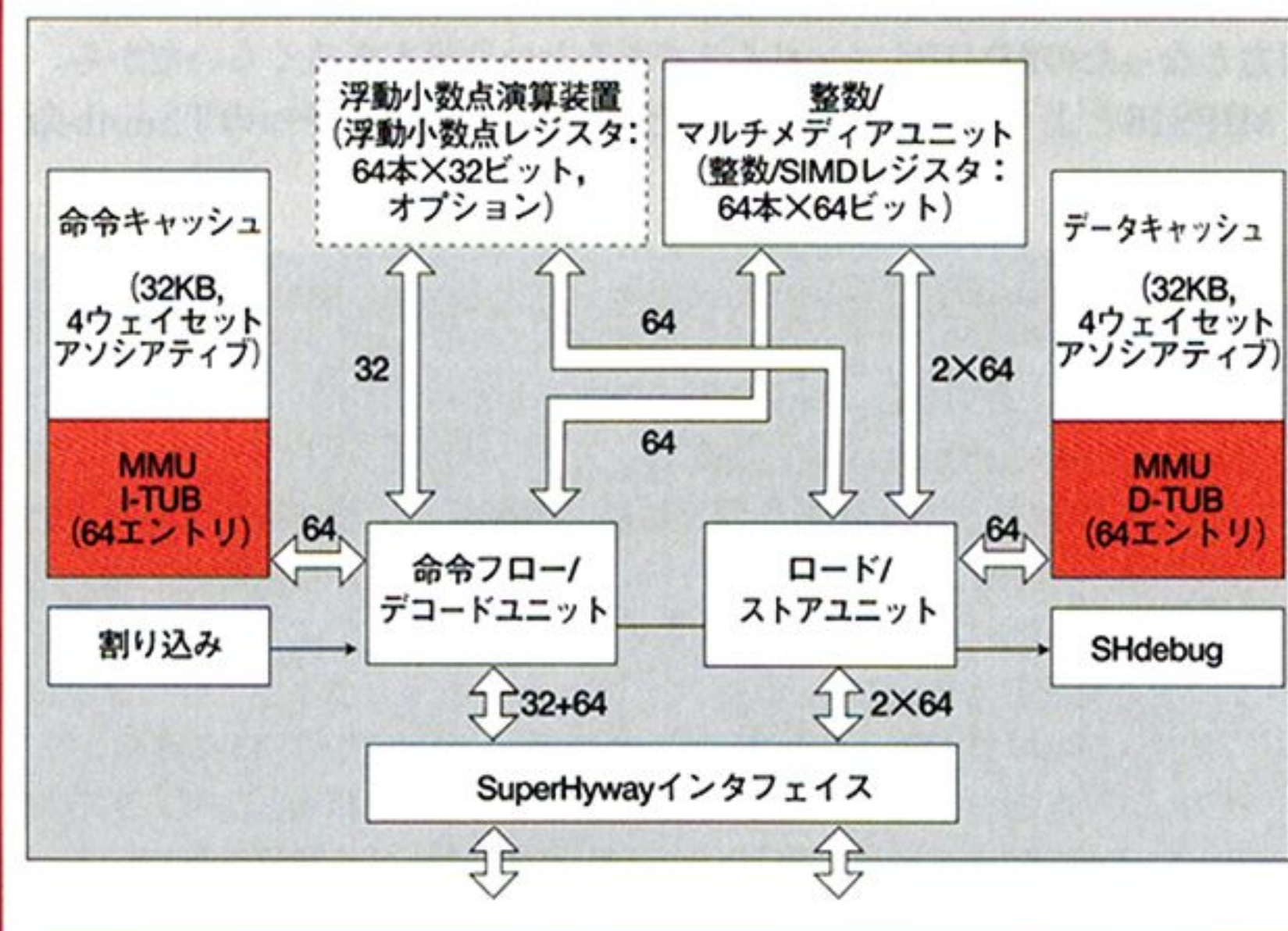
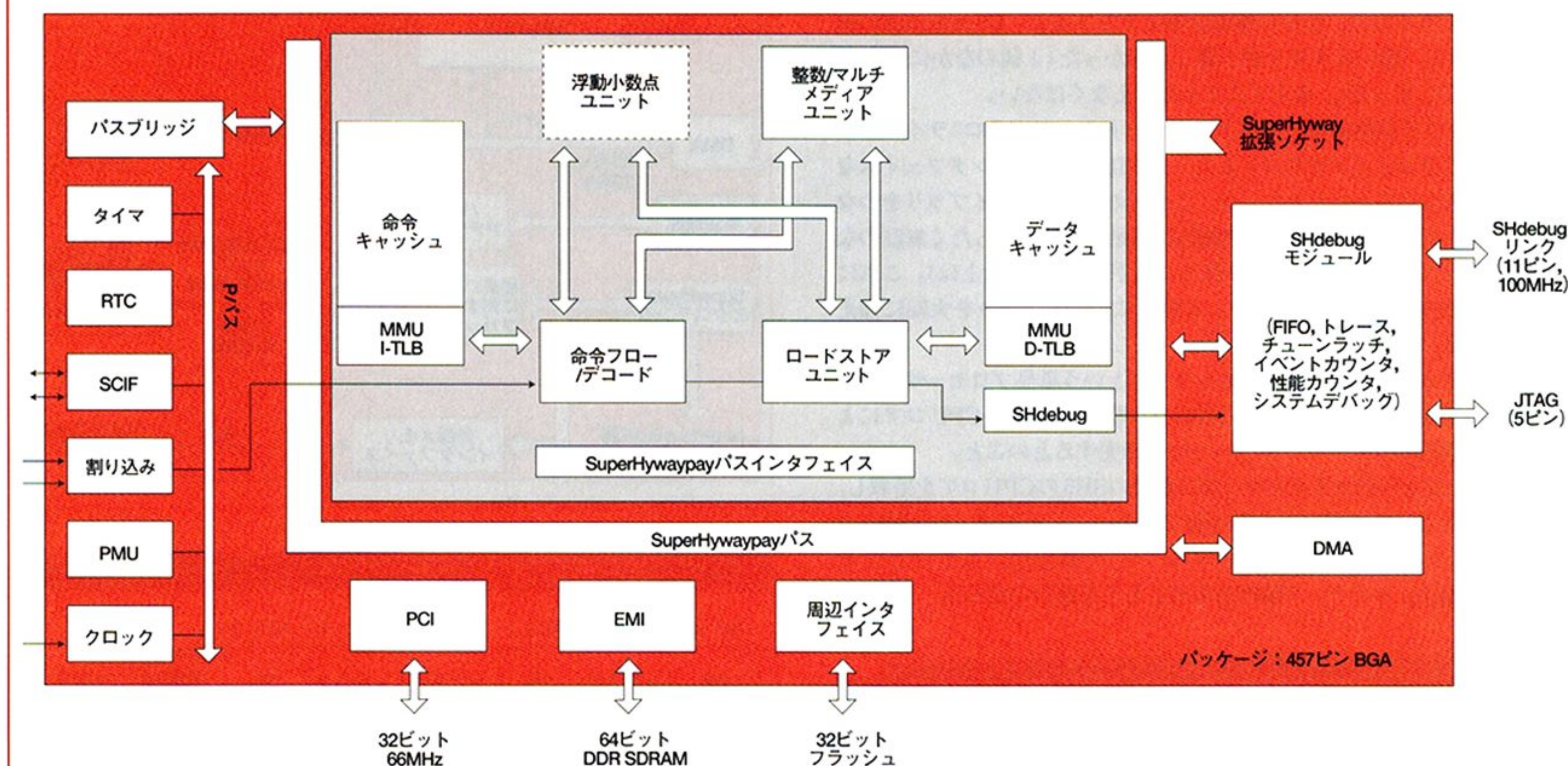



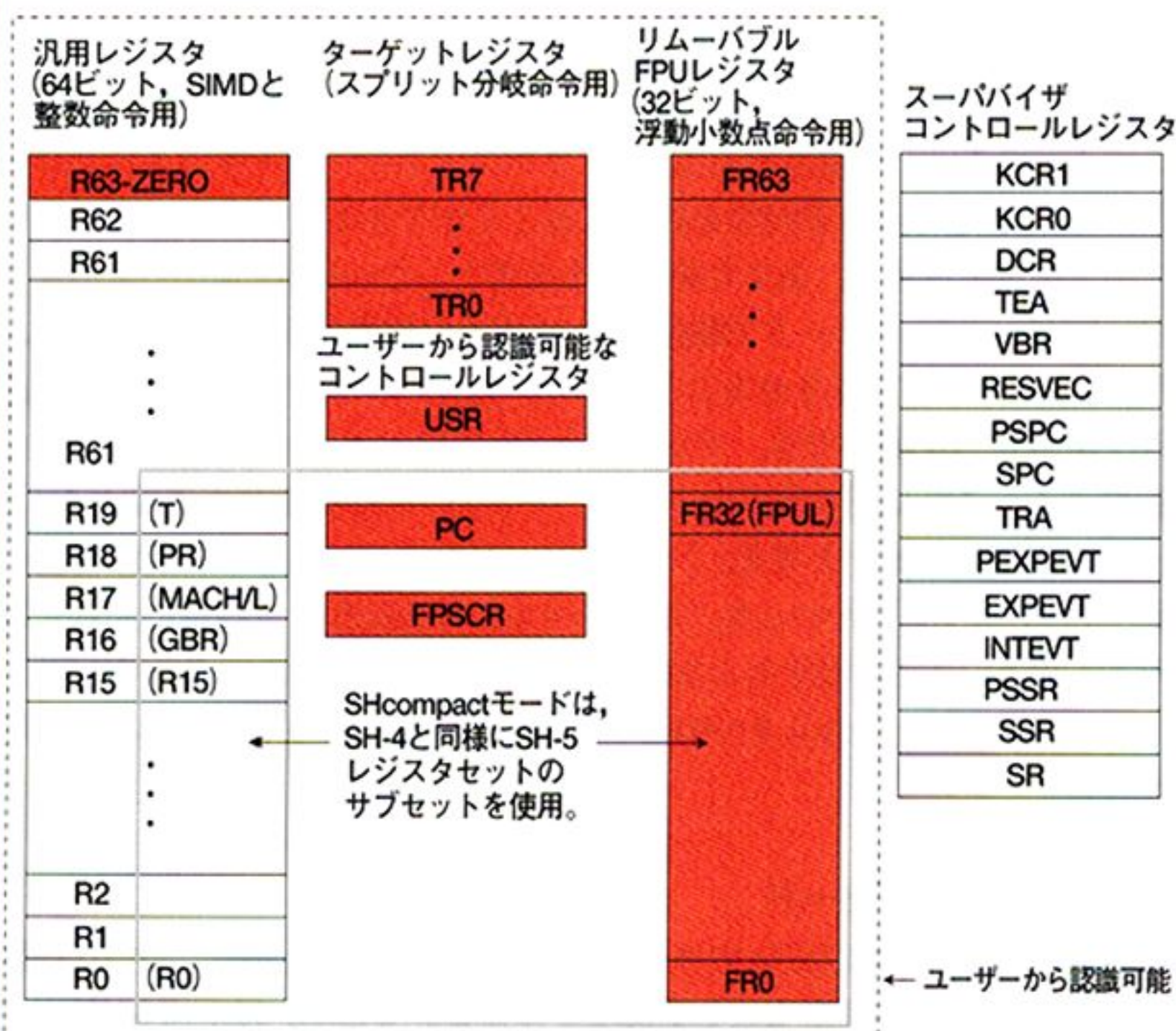
図2　SH5アーキテクチャを採用した最初の製品の構成

SH4の32ビット2wayスーパースカラは，32ビットのデータ操作を2つ同時に処理できる（命令を処理できる箇所が2つある）。対してSH5の64ビットシングルスカラは，64ビットのデータ操作をひとつずつしか処理できない（命令を処理できる箇所はひとつしかない）。つまり，両者とも同時に処理できるビット数は64である。一般に，整数演算においては，64ビットより32ビットのデータのほうがよく扱われるため，32ビット×2でデータを処理できるSH4のほうが効率よくデータを処理できそうに思える。ところがドッコイ，SH5は64ビットのデータを32ビット×2とみなして一度に処理する命令もある。少々ヤヤコシイが，SH4の場合，32ビット×2のデータを2つの命令処理箇所がひとつずつ受け持つコトで，1単位時間で32ビット×2のデータ処理ができる。対して，SH5の場合，32ビット×2（=64ビット）のデータをひとつの命令処理箇所が2つ同時に処理することで，1単位時間で32ビット×2（=64ビット）のデータ処理ができるというわけなのだ。

スーパースカラはその効果も大きい分，回路規模も大きくなり，かつ非常に複雑になるという欠点を持つ。シンプルな回路構成で高速実行をウリにしているRISCプロセッサとは正反対の特徴があることからもわかるように，スーパースカラはえらく実装難度の高い，高速化の最終兵器ともいえる。逆に，基本構造を変えたばかりのSH5にスーパースカラが搭載されているのは，ある意味おかしく「スーパースカラの前にやることはいくらでもあるだろう」ということといえる。

●必要/不必要が分かれるからオプション

意外というかウマイというか，IEEE-754倍精度FPUはオプションとなっている。ご家庭用向き制御にFPUを標準装備する必要があるか？　というように，次世代ゲーム機用プロセッサとしてではなく，純粋なマイコンとしてのSH5の位置づけを考えれば十分に納得がいく。

さて，このオプションのFPUは，32ビットレジスタを64本持っており，内積演算（ポリゴンの表裏判定など）やマトリクス演算（アフィン変換：拡大・縮小・回転など）を1サイクルで実行できる。いまとなってはさほど驚きはないものの，浮動小数点演算においても，その並列性は非常に高い。また，オプションという位置づけにもある種の期待を感じずにはいられない。早い話が，オプションのFPUだけ独立進化するという可能性である。

図3　SH5 CPUレジスタセット

SOC（システムオンチップ）を実現するマイクロプロセッサコア

ある意味SH5の最大の目玉であるのが，「SH5のCPUコアはハードウェアマクロとして提供」されるということ。これまでは聞き覚えのある単語ばかり並んでいるのだけど，ここへきてサッパリなんのことだかわからない，なんて人も多いのではないだろうか。

時代は進むモノで，最近はその動作をソフト（ハードウェア記述言語）で記述するだけで，いわゆる「カスタムチップ」（X680x0でいうところのシンシアなど）を作ることができてしまう（ASIC（エイシック）だとかVHDL（ブイエイチディーエル）といった単語をよく耳にするハズ（編注：普通の人はしないと思う））。SH5のCPUコアは，このようなASICモジュール，「SH5

のCPUとして動作するようハードウェア記述言語で書かれたマクロ」として提供される。つまり，「SH5入りのカスタムチップ」を（比較的）簡単に作成することができるのだ。かなり極端な例え方をすると，Dreamcast2（仮名）を分解したら，黒いカタマリが1個しかなかった（1個のなかにSH5ほかを全部入れてしまった）。なんてこともありえなくはない。

SH5のCPUコア以外にも，さまざまなハードウェアマクロ/ライブラリが存在する。フラッシュメモリコントローラやIEEE1394インタフェイスなど。気づいた人もいるかもしれないが，これらのマクロやライブラリをつなぎあわせることで，本当に必要な「機能的に満足のいく，まったく無駄のない」たったひとつのカスタムチップを作ることができる（理論上は）。これにより，部品点数を筆頭に，ハードウェア開発にかかるコストを大幅に抑えることができる。

SH5は，従来どおり日立のSH8000シリーズという単品プロセッサとしても発売される予定である。SH8000には日立が自前でSH5のCPUコアによく使いそうな機能のライブラリをひっつけて発売するとのこと。

そうなると，ひとつのカスタムチップに複数のSH5のCPUコアを搭載し，同時に動作させる，なんてコトも不可能ではない。たとえば，2つSH5コアを搭載し，ひとつはDreamcastのコンパチ動作用，もうひとつをDreamcast2（仮名）ネイティブ動作用などとしても面白いだろう。

2種類の動作モード

SH5は，2つの動作モードを持つ。ひとつ目は，SH4のユーザーモードと互換のある「SHcompactモード」。これまでのSHシリーズ特有の16ビット固定長命令コードを持つ，互換性やコード効率を優先したモードといえる。そして，2つ目が，「SHmediaモード」。これまでのSHシリーズと異なる32ビット固定長命令コードを持ち，SH5で拡張されたレジスタにアクセスすることができ，高度な演算命令や並列演算を処理できるようになっている。

●混在の意味

このようなモードの混在からも「マイクロコントローラ＋マイクロプロセッサ＝SH」ならではの方向性を感じ取ることができる。つまり，ソフトウェアの大部分を占める，あまり演算力を必要としない部分（エラーチェックなど）をSHcompactモードで動作させ，「本当に演算力が必要」なときのみSHmediaモードで動作させることにより，実行パフォーマンスをさほど落とさず消費電力を節約でき，プログラム（オブジェクト）コードのサイズも節約できる。これは「『ひたすらフルパワー』はいろんな資源を消費する」という現実問題をキチンと考え，「バッテリー長持ち」な製品や「ROM容量の節約（コストダウン）」といった，現場レベルでの製品作りを視野に入れた設計そのものといえる。出すところは出し，締めるところは締める。実に魅力的といえる。なお，これら2つのモードは分岐命令によりダイナミックに切り替わる。

●SHcompactモード

SHcompactモードはSH4のユーザーモードと互換性があり，計201種類の16ビット固定長の命令セットを使用することができる。ほかにもSHシリーズならではの，2オペランド命令，16本の32ビット汎用レジスタももちろん健在である。ユーザーモード互換というトコロが，まぁ，これまで，16ビット固定長でやってきたツケと思えなくもないが，しかたないことだろう。逆に，ユーザーモード互換とすることで，ここまでの互換性を維持できたのだろう。

●SHmediaモード

SHmediaモードは，201種類の32ビット固定長の命令セットを使用する。命令書式も3オペランドとなった。整数演算，SIMD命令，浮動小数点演算（IEEE-754オプション）はもちろんのコト，どう考えても，ポリゴン用命令，デジタルオーディオ用命令にDVD用命令などと，まんま，次期情報家電の中心を狙っているとしか思えない構成が，実に印象的である。汎用レジスタもレジスタバンクなどというコスい技を使うことなく，ごく自然体な64本の64ビットレジスタと，大幅に改善された。

図4　SH-5を使用したSOCの例

図5　デュアルモード命令セット

図6　SAD演算

SAD:Sum of Absolute-Deference，差の絶対値の合計

図7　スプリットブランチ命令の実行

命令キャッシュ
PTA/L #loop, R0 ←ターゲットアドレスの準備（予測付き：Ready）
MOVI #20, R0 ←ループカウンタのロード
Loop: instruction 0 ←ループ内の命令
instruction 1 ←ループ内の次の命令
ADDI R0, #1, R0 ←ループカウンタのデクリメント
BNE/L R0, #0, TR0 ←テストおよび分岐（予測付き：Ready）

プログラムカウンタ
フェッチバッファ0
フェッチバッファ1
フェッチバッファ2
フェッチバッファ3
ターゲットバッファ
命令0
命令1
ターゲットレジスタ
TR0
ターゲットレジスタ8個
命令X
命令Y
TR7
マルチプレクサ
実行パイプライン

実行結果のトレース*
n PTA/L ploop,R0
n+1 MOVI p20,R0
n+2 Instruction 0
n+3 Instruction 1
n+4 ADDI R0 #1,R0
n+5 BNE/L R0,#0,TR0
n+6 Instruction 0
n+7 Instruction 1
n+8 ADDI R0 #1,R0
n+9 BNE/L R0,#0,TR0
n+10 Instruction 0
n+11 Instruction 1
n+12 ADDI R0 #1,R0
n+13 BNE/L R0,#0,TR0
n+14 Instruction 0
n+15 Instruction 1
n+16 ADDI R0 #1,R0
n+17 BNE/L R0,#0,TR0

*キー：
フェッチバッファから
ターゲットバッファから

実行結果トレースの説明
n:
PTAはIキャッシュからデータをプリフェッチし，ターゲットバッファへ格納
n+2/n+3:
フェッチバッファからの命令0と命令1をデコード
n+6/n+7:
ターゲットバッファからの命令0と命令1をデコード
n+10/n+11:
ターゲットバッファからの命令0と命令1をデコード
n+14/n+15:
ターゲットバッファからの命令0と命令1をデコード

図8　フルフォワーディングの例

ADD R1,R2,R3	F1	FD	D	E1	E2	E3	W		
ADD R3,R5,R6		F1	FD	D	E1	E2	E3	W	
ADD R6,R8,R9			F1	FD	D	E1	E2	E3	W

（矢印はフォワーディングを示す）

この64ビットのレジスタ。先にも少し触れたが，命令によっては8ビットデータ×8，16ビットデータ×4もしくは32ビットデータ×2として一度に扱われ，さらにこれらのデータを1命令でまとめて演算できたりもする。具体的な例（というよりSH5の目玉命令）を見てみよう。まず8ビットのデータが8つパックされた64ビットのレジスタが2つあるとする。仮にレジスタをαとβとし，それぞれにパックされている8ビットデータをA0～A7およびB0～B7とする。

α：A7 A6 A5 A4 A3 A2 A1 A0
β：B7 B6 B5 B4 B3 B2 B1 B0

この状態で，SAD（Sum of Absolute Deference）という命令を実行すると，パックされた8ビットデータ同士の差の絶対値の合計値が得られる。もう少し冗長に説明すると，

(A7-B7)の絶対値＋(A6-B6)の絶対値＋(A5-B5)の絶対値＋(A4-B4)の絶対値＋(A3-B3)の絶対値＋(A2-B2)の絶対値＋(A1-B1)の絶対値＋(A0- B0)の絶対値

が1命令で得られる。つまり，1クロックサイクルで8回の減算，8回の絶対値算出，8回の加算が実行される。う～む，命令の並列性は，非常に高いとしかいいようがない。気づいた人もいるかもしれないが，例に挙げたこの命令，思いっきりMPEGのコード化に必要な演算そのものだったりする。やはり，次期情報家電の中心を狙っているからこその命令といえよう。ほかにもSIMD命令は，符号付き/符号なし小数演算，および飽和/モジュロ演算を実行できる。

高速実行するためのあの手この手

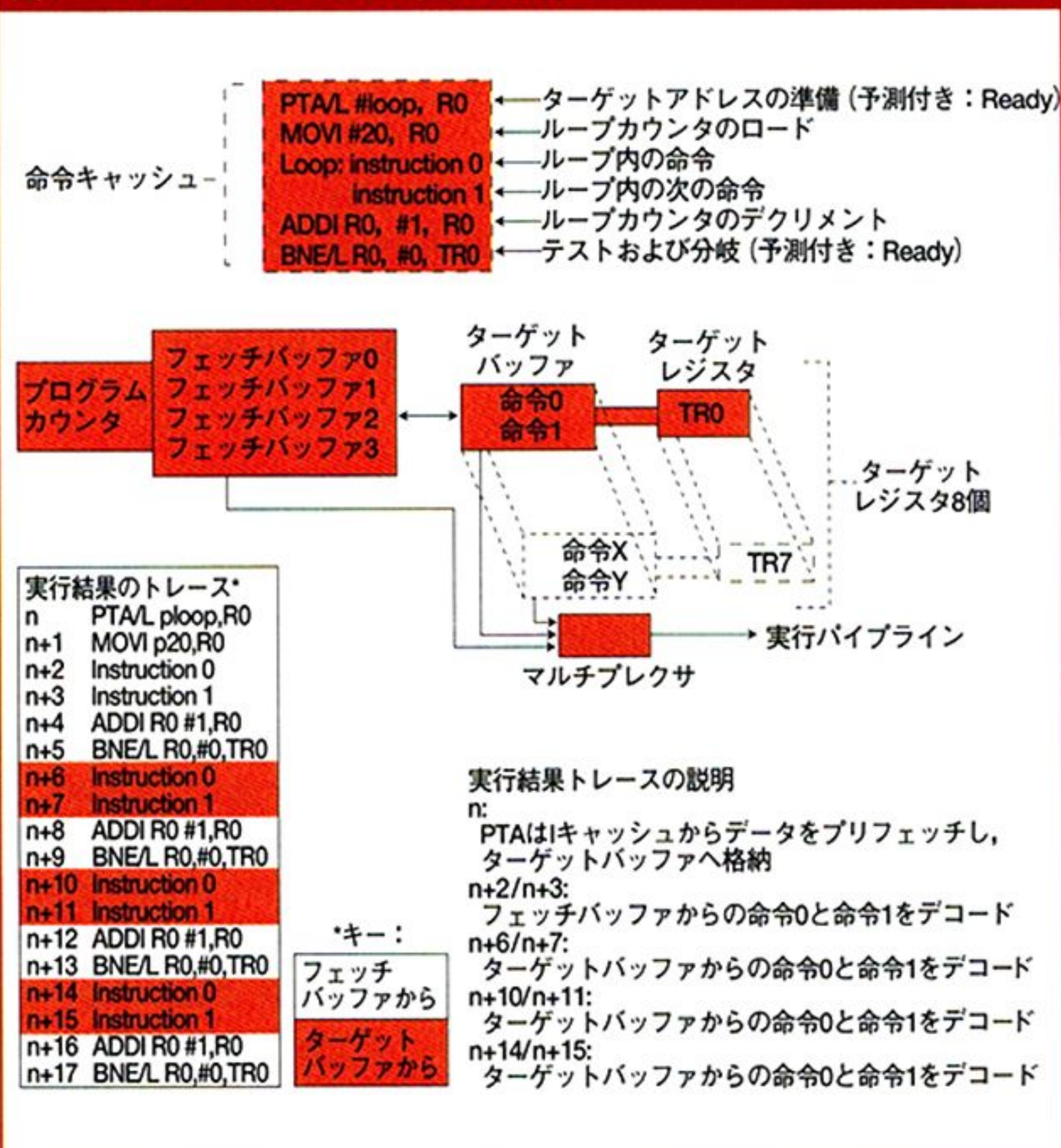
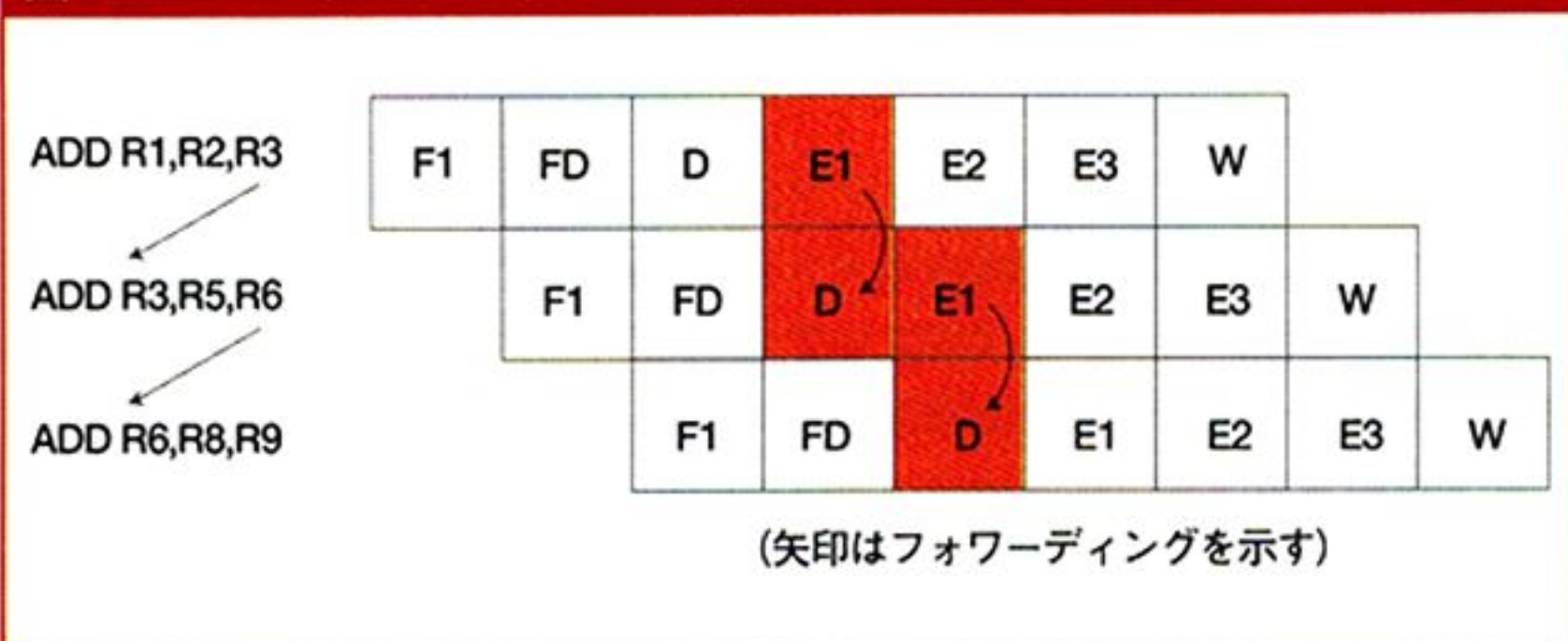
SH5には単なる実行速度の向上はもちろん，その並列性を最大限に活かすために，文字どおりあの手この手を使っている。

●スプリットブランチアーキテクチャ

「分岐するかもしれないアドレス」をあらかじめ設定することで，分岐時の時間的ロスを抑えることができる，という説明が的確かつ簡単だろう。忘れてしまった人，もしくはそもそも知らない人もいるかと思うので，一応，説明しておくと，単純であろうと複雑であろうと「ループ＝分岐」でなのだ。また，ループの内側の処理が軽ければ軽いほど，分岐処理の時間は深刻な問題となる。条件分岐であろうと無条件分岐であろうと，関係ない。実際，CPUにM68000を搭載したX68000においては単純ループにおける分岐時間は敵ですらあり，ループの展開はアセンブラプログラマのたしなみであった。ほかにも，Cコンパイラにもループを展開するためのオプションが用意されており「分岐予測」にかなりのウェイトを置いているプロセッサもよく見かける。と，少々ハナシがそれてしまったが，本題に戻ると，

```
sum=0;
for (z=1 ; z<=9; z++){
for( y=1; y<=9; y++ ){
for( x=1; x<=9; x++ ){
sum+=(x * y * z);
}
}
}
```

というようなCのプログラムをコンパイルしたとき，その見かけからは想像もつかないほど数多くの分岐処理を行う。「出力されるアセンブリ言語のソースの質」という問題もあるが，この場合，演算時間より分岐処理に時間を取られてしまい全体的な実行速度の足を引っ張ってしまう。

ではなぜRISCプロセッサは分岐を苦手とするのか？　という問題であるが，それは，RISCの高速性のキーである命令パイプラインが分岐と同時にクリアされリフィルに時間がかかってしまうからなのだ。

RISCプロセッサ（の命令パイプライン）はよくベルトコンベアの流れ作業に例えられる。まさにそのとおりで，RISCプロセッサは「都合のよい状態で，順番よく並んでいる命令を次々に実行する」コトに関してのみ，プロフェッショナルの仕事を見せてくれる。分岐処理とは，いわばそのベルトコンベアの上にのっている命令を一度全部降ろし（クリア），新しい命令を1から積み直す（リフィル）作業といえる。ベルトコンベア上の作業であるから，積み直したあとは普段どおりの速度で動くものの，「降ろす＋積み直す」作業には時間が取られてしまう。

そこでSH5は「分岐するかもしれないアドレス」をあらかじめ設定することで，実際に分岐する前に，分岐直後に実行すべき命令を解釈しておく。これにより，分岐命令によりクリア/リフィルの動作をしながらも，同時（「本当に分岐命令直後」のタイミング）に，分岐先の命令を実行することができる。つまり，あらかじめ解釈しておいた分岐先の命令を実行することで，見かけ上，パイプラインのクリア/リフィルの時間を隠すことになるのだ。

●フルフォワーディング

SH5の命令パイプライン，および命令ステージは7段ある。

- フェッチ1(F1)
- デコードフェッチ(FD)
- デコード(D)
- 実行1(E1)

・実行2(E2)
・実行3(E3)
・ライトバック(W)

パイプラインは切り離されたパイプファイルを使い，ライト結果をライトバック段階でセットされたレジスタに書き戻す前に，その結果をストアする。これによりフルフォワーディングが可能となる。パイプライン化されたバックツウバックのMAC命令とパイプライン化されたストアをサポートしている。

たとえば，

```
ADD R1,R2,R3      ;R1+R2→R3
ADD R3,R4,R5      ;R3+R4→R5
ADD R5,R6,R7      ;R5+R6→R7
```

を実行した場合，通常，1行目の「ADD R1,R2,R3」の命令において，実際にR3レジスタにR1レジスタとR2レジスタの値を足した結果が格納されるのは，ライトバックステージとなる。したがって，R3レジスタの値を使用する2行目の「ADD R3, R4, R5」は，1行目のライトバックステージの完了を待ってからでないと実行できない。……というのがRISCプログラミングの常識であったのだが，SH5においては，上記の工夫(技術)において，R3レジスタへのライトバックを待つことなく，2行目の「ADD R3,R4,R5」が実行されてしまうようになったのだ。そのようなワケで，同様に2行目の演算結果を必要とする3行目も，2行目のライトバックを待つことなく実行できてしまうので，結果，1～3行の間は，待ち時間なしの，CPUパワー全開で実行される。

●論理キャッシュ

CPUコアには，4ウェイセットアソシアティブ(32ビットキャッシュライン)のそれぞれ32KBの論理命令キャッシュと論理データキャッシュを備えている。それぞれに64エントリのフルアソシアティブトランスレーションルックアサイドバッファ(TLB)が用意されている。どうでもよいが長い名前だ。

論理キャッシュは，キャッシュミスがない限り，TLBを参照せずにキャッシュアクセスできるため，高速動作そして低消費電力性に優れている。また，論理キャッシュにより，データのスループットの向上やTLBのミスの抑制にもつながるとされているが，このあたりは文字どおり使い方次第といったトコロだろう。また，動作モードの切り替えにおける影響がどの程度出るかなど，実物が出てこないことにはなんともいえない部分もある。

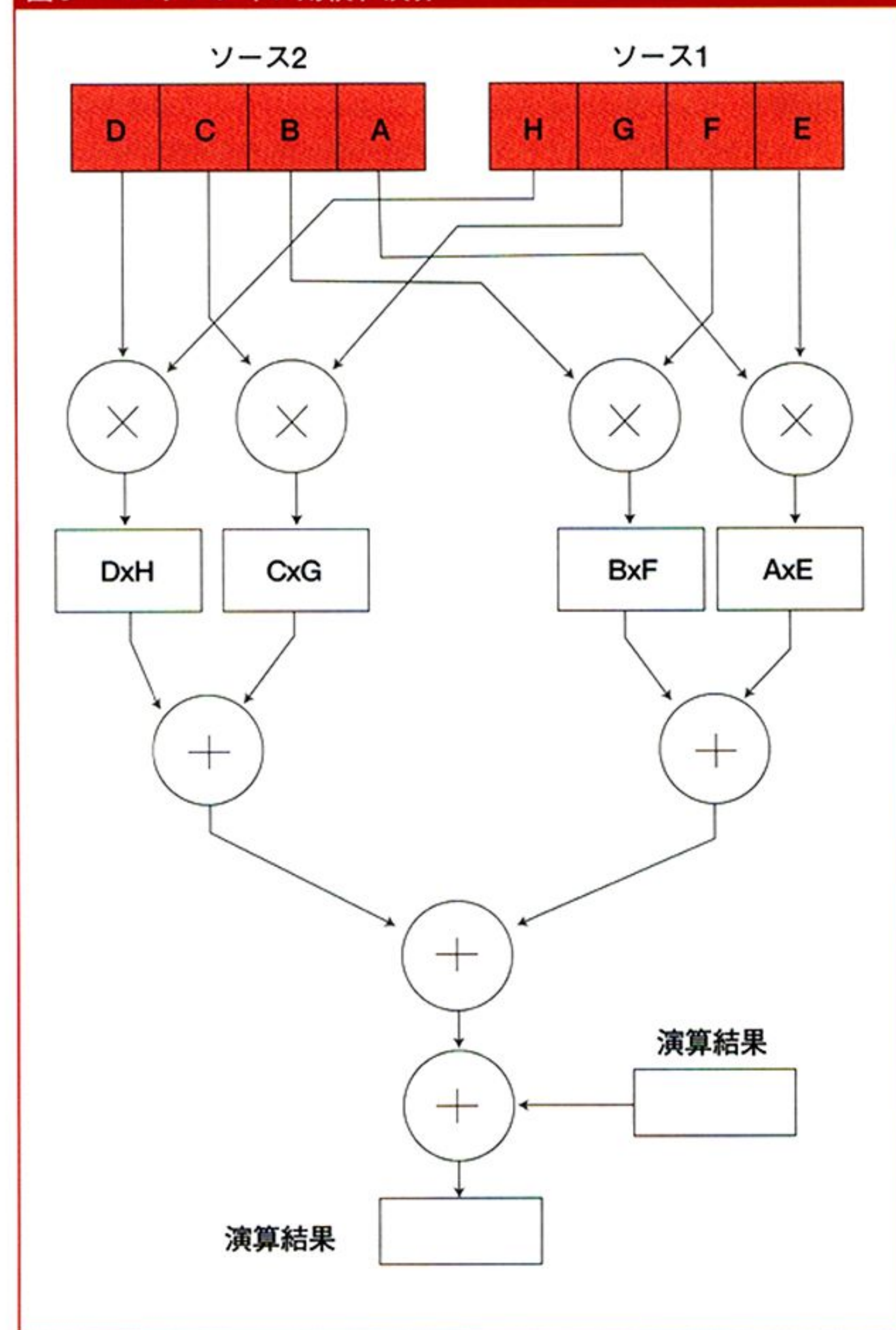
なお，特権モードとユーザーモードで双方のキャッシュをロックできるので，心おきなく「ブン回すことを前提にしたコード」や「キャッシュから外れてほしくないデータの確保」もうまく記述できるだろう。

●マルチメディアユニット

このなんともトホホな名前のユニットは，整数演算ユニットと汎用レジスタを共有しているため，余分なユニット間のデータ転送の手間が省ける。先にも触れたように，64ビットの汎用レジスタはSIMD命令において，8個の8ビットデータ，4個の16ビットデータ，2個の32ビットデータのパックとして扱われる。

それにしても，ユニットの名前はともかく実用本位というか「実際に組み

図9　マルチメディア用積和演算

ソース2
D C B A
ソース1
H G F E
DxH
CxG
BxF
AxE
演算結果
演算結果

分岐について　COLUMN

たとえばCにおけるループ処理の記述は，

```
int sum,i;
sum=0;
for( i=1; i<=10; i++ ){
        sum+=i;
}

int sum,i;
sum=0;
i=1;
while(i<=10){
        sum+=i;
        i++
}
```

このように，for文やwhile文を使用するが，結局のトコロ，コンパイラは，

```
int sum,i;
sum=0;
i=1;
loop_top:
if( i<=10){
        sum+=i;
        i++;
        goto loop_top;
}
loop_end:
```

というような，if文(条件判定/条件分岐)とgoto文(無条件分岐)による組み合わせのループ処理に置き換え，それぞれのターゲットCPU用のアセンブラソースを吐き出す。

通常，RISCプロセッサにおいては，上記の例でいうところの「goto loop_top;」においてパイプラインがクリアされるため，実際の制御がloop_topに到達(リフィルが完了)するまでのタイムロスは避けられない。SH5のスプリットブランチアーキテクチャは「loop_topに分岐する」ことがあらかじめわかっているのであれば「loop_top直後の命令をあらかじめ取り込んでおく」ことで，パイプラインのリフィルによるタイムロスをなくせるというもの。loop_topの具体的なアドレスは，コンパイラがはじき出すようにすれば問題ない。そういう意味では，スプリットブランチアーキテクチャはずいぶんとソフトウェア寄りのアーキテクチャといえる。

込まれる製品でなにを行うか」という点をしっかりと考えている命令セットは実に頼もしい。先に紹介したSAD命令はMPEGのコード化用命令そのものであるし，たとえば，MMULSUM（マルチメディア用積和演算）命令は16ビットオーディオ（CDやMD）用インパルスフィルタ用命令そのものである。

それこそ，４段のFIRフィルタであれば，フィルタ係数を16ビット固定小数点で，レジスタ（64ビット=16×4）にパックし（実際にはインタリーブ処理が必要だが，ここでは省略）CDから読み込んだデータをレジスタにパックすれば，MMULSUM命令一発でデジタルフィルタ演算は完了。あとは，演算結果（32ビット）の上位16ビットを取り出してオーディオ出力するだけ。これだけ多くの汎用レジスタを持ち，これだけ高い並列演算性能を持っているのであれば，多次室内音場シミュレーションから高速N点平均算出まで，いろいろと面白いことができる予感。

●ポリゴン用（?）浮動小数点ユニット

オプションとなっている浮動小数点ユニット（FPU）は，64個の32ビット浮動小数点レジスタを持ち，倍精度演算もサポート。パイプラインは９段。４クロックサイクルで，４次のマトリクス演算が可能。400MHz動作時に2.8GFLOPSの性能となる。

……と確かに何年か前では想像もつかないスペックではあるものの，EmotionEngineのあとでは，それほど感動がないという人がほとんどだろう。

●SuperHywayバスなど

VSI（バーチャルソケットインタフェイス）プロトコルと互換性のある64ビットバス。物理的に２本のリードライトバスで構成され，完全な全二重通信方法（要求/応答の同時転送）をサポート。バンド幅の最大値は3.2GB/sec（バスクロック200MHz時）。SuperHywayバスは，要求をパイプライン処理できるため，待ち時間の長いモジュールも使用でき，PCI周辺機器に対しては物理アドレス空間のキャッシュスヌーププロトコルをサポート。

プロセスには，0.15μm６層銅メタルCMOS技術を使用。

割り込みレベルは16段階で，NMI可能。割り込みとTLBミスはそれぞれ独立したオフセットを使用。

情報家電の価値とは

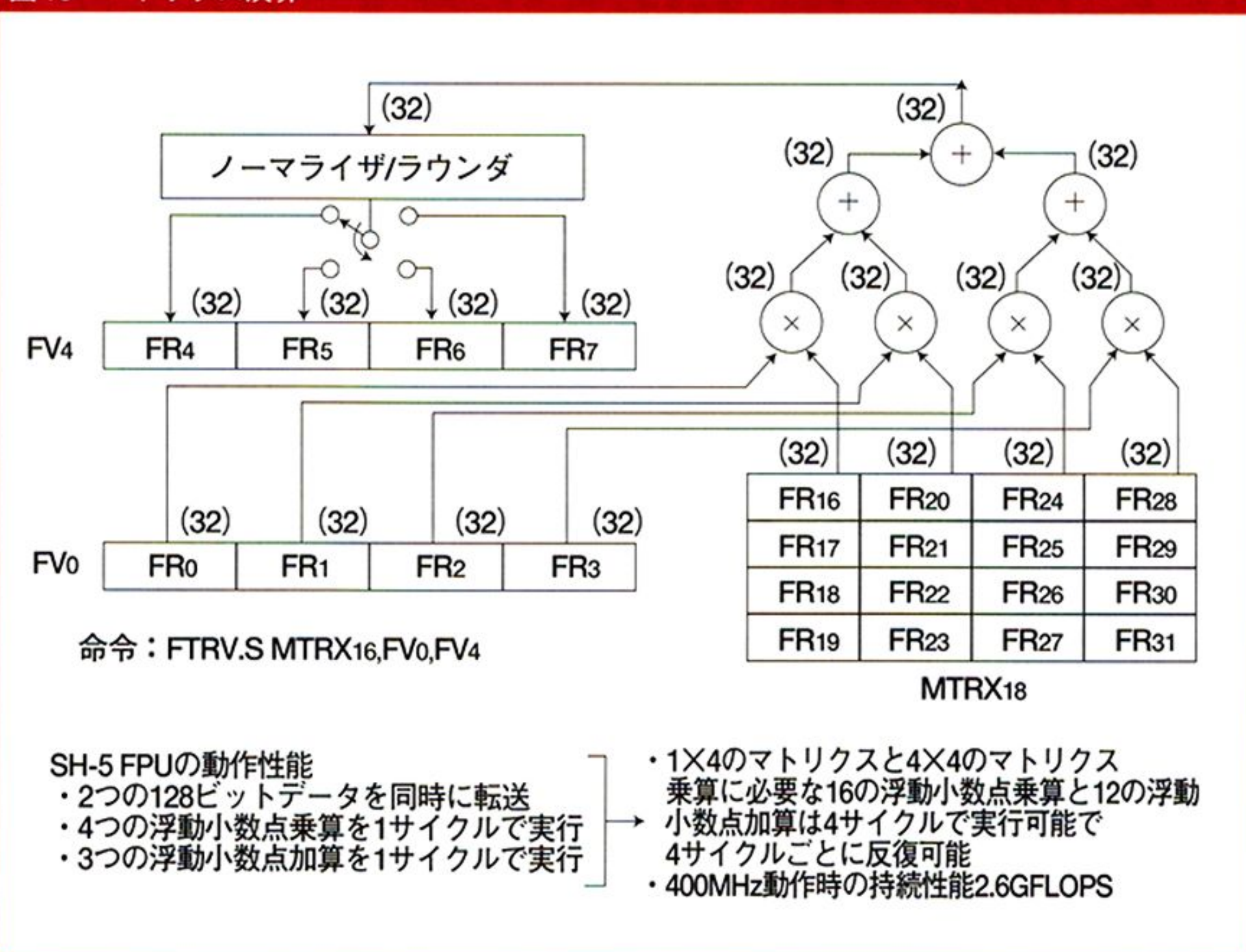
まだ姿を見せていないものの，SH5はどうも「ゲームも視野に入れた」マイコン。というニオイが強いように感じる。そういう意味では，SH4のときのような，「ゲーム用プロセッサで天下とるでぇ」というようなハナイキは，少し落ち着いたようにも感じる。

それでも，SOCというアプローチをとったり，FPUをオプションにするなど，いろいろと「日立の野望～全世界版～」な考えが見え隠れするのは面白い。ほかにも「1024レジスタのバタフライアクセスユニット」や「ジグザグ64配列演算ユニット」なんかを用意して，ハッタリの利いたハードウェアライブラリの充実を願いたい。

しかし，本気でSOCのキープレイヤーを狙うなら，SH5のCPUコアだけでも，限りなくタダの状態でバラまいてみるのもいいかもよ？　日立さんってねえ。

図10　マトリクス演算

命令：FTRV.S MTRX16,FV0,FV4

SH-5 FPUの動作性能
・2つの128ビットデータを同時に転送
・4つの浮動小数点乗算を1サイクルで実行
・3つの浮動小数点加算を1サイクルで実行
→
・1×4のマトリクスと4×4のマトリクス乗算に必要な16の浮動小数点乗算と12の浮動小数点加算は4サイクルで実行可能で4サイクルごとに反復可能
・400MHz動作時の持続性能2.6GFLOPS

パイプラインは詰めてナンボ　*COLUMN*

これまでのRISCプログラミングにおいては，「パイプライン詰め」という基本技（？）的たしなみ（？）があった。と同時に，これがキチンとできているかどうかで，プログラマがプロセッサの特性をつかんでいるかどうかをある程度判断できた。たとえば，SH2における，

```
ADD.L R0,R1     ;R1=R1+R0 (※1)
ADD.L R1,R2     ;R2=R2+R1 (※2)
MOV.L R0,R4     ;R4=R0 (※3)
```

という処理の場合，※２の行は，R1レジスタの値，つまり※１の行の演算結果が，実際にレジスタに格納されるのを待ってからでないと実行できない。※２の行のパイプラインが実行ステージにあるとき，残念ながら，※１の行のパイプラインは演算結果の格納を終了できていない。つまり，※２の行において「待ち」が入る。見かけ上，３クロックですむべき処理が４クロックかかってしまう。つまり，パイプラインがスムースに流れていないのだ。

そこで，演算結果が実際に格納されるまでの間に「演算結果格納待ちをしていない」かつ「論理的に矛盾しない」処理を挿入する，という考えになる。具体的には，※３の行を※２の行の前に移動する（入れ替える）だけなのだが。ともあれ，実際にソースを見てみよう。

```
ADD.L R0,R1     ;R1=R1+R0 (※1)
MOV.L R0,R4     ;R4=R0 (※3)
ADD.L R1,R2     ;R2=R2+R1 (※2)
```

順番を入れ替えた※３の行では演算結果の格納待ちをしていない（する必要がない）ので，待つことなく実行される。そして，※２の行を実行する頃には，※１の演算結果の格納は完了している。よって，この行も待つことなく実行される。結果，プログラム的にまったく待つことなく，また時間的な無駄（ロス）もない。このように，プロセッサが効率よく処理できるようにソフトウェアを記述することを「パイプラインを詰める」という。逆のいい方をすれば，プロセッサがどのようなタイミングで動作しているかをきちんと理解していなければ，パイプラインを詰めることはできない。

ほかにも，SH2（7604）の場合，乗算命令の結果がMACレジスタに格納されるまでに数クロック，外部除算機による演算完了までに30クロックなど，ただ演算を待つだけでなく，待っているあいだにできる作業があればドンドン先に進めて全体の実行効率を上げる。

と，いうトコロまでが，おさらい。最近は「シンプルisベスト」がモットーのRISCプロセッサといえども状況が変わり，プロセッサそのものがパイプラインの流れをよくしようと，あの手この手を尽くしている。SH5においては，フルフォワーディングにより，RISCプロセッサであるにもかかわらず，上記例のような「パイプライン詰め」の必要がなくなった。SH4における「0サイクルレイテンシ」（ごく簡単な命令に関してはデコードステージにおいて実行が完了するという仕組み）にも驚いたが，この進歩の早さにはもっと驚かされる。

ネットワーク対応3Dゲーム
Galaxy-Knightsの制作（中編）

サイバーヘッド

2000年春号に引き続き，DirectXを使用したネットワーク対応3Dゲームを制作していきます。今回はDirectX8に準拠してプログラムを全面的に改訂しました。DirectX8プログラマの方は参考にしてください。

最新動向1：業界の流れ

長い，なが〜い時間を経ての第2回です。いや本当に長いですね，ちょっとした市販ゲームを1本作れる時間が経ってしまいました。世間の動きも激しく，PS2が発売となり，Xboxが発表され，GAME-CUBEもお目見えしました。ゲームも次の次元に進む準備が，着々と進んでいるようです。対極に位置する携帯電話もJava対応になり，いよいよゲームの活動する分野が広がってきたかに見えます。

ですが，現実を見てみると，日本のゲーム業界の寂しさが気になります。2000年初頭にも懸念はしていたのですが，中小企業が相次いで倒産＆撤退し，大企業ですら倒産する厳しい冬の時代が到来しています。さらに，少数残ったゲーム市場はさまざまな要因からいよいよ大資本中心となり，かつてゲームを支えてきた中小のゲームソフトハウスが，生きていくのが難しい時代になってしまったようです。

これは，ゲームという商品の性質を考えると，あまり喜ばしい時代ではありません。大資本が中心となってアーティストや下請けの会社に仕事を与えていくというのは，資本主義の図式としてある面では正しいのでしょうが，そこには大資本の囲い込みにより，中小の会社が没落する図式があります。

独自で仕事を受けるには資金が足りない→
大手のプロジェクトの部分を引き受ける→
大手は自社で作業をしてもらいたい→
中小企業の社員が大手に出向→
出向した人間は大手の環境に慣れてしまう→
慣れた環境のある大手に移籍→
中小企業が弱体化する。

淘汰の原理が働いているのは確かですが，実はこれは，業界全体の弱体化の始まりでもあるのです。どういうことかというと，大資本と中小企業では以下のような違いがあるためです。

	大資本	中小企業
広告・製造力	マスメディア+大々的展開（10億円）	店頭レベルのみ（3千万円）
設備投資	きわめて大きい（5億円）	必要最小限（5千万円）
制作人員	数十〜数百人（10億円）	10人前後（6千万円）
制作期間	数年	1年前後
制作費回収最低本数	100万本以上	6万本前後

括弧内の数字は経費です（実際には製品の質により大幅に変わりますので，あくまで仮と考えてください）。上から4つは，明らかに大資本が勝っていることを示しています。それはそうでしょう。中小企業と違い，投資の幅がけた違いなのですから。問題はそれから導き出される結果です。

たとえば，通常のゲームの場合，売り上げは製品価格の4割程度と考えてよいでしょう。だいたい6,000円として，1本あたり2,400円程度になります。そこから求めた数値が最後の回収最低本数です。実際には，上記のうち設備投資が1回きりということはないので，その分は浮きます。

つまり，概算（どえらく丼勘定ですが）の必要経費が中小企業は1.4億円，大企業は25億円とすると，回収には数十倍の本数が必要となります。ところが，資金をつぎ込めばその分確実に売れるかというと，実際にはそうはいきません。5万本売れるソフト自体がなかなか出てこないのが現実です。大資本がバンバン宣伝を打っても，100万本売れるソフトは年間10数作前後。ましてや，ソフトは販売してからしか資金が回収できません。いくら資本力があるといっても，年間10本以上作品を出していく大資本は，数百億以上の出費を余儀なくされるわけで，ヒット作がなかなか出ない状況が続くとあっという間に経営危機に陥ってしまいます。つまり，大資本は「常に大ヒットを出す必要」に苛まれることになります。近年，大手の和議申請があい次いだのがすべてを物語っているといえるでしょう。

それに対して，中小企業はその気になれば1作1億未満で作ることも可能です。販売数2万〜3万本で資金回収することも夢ではないわけで，そこそこちゃんとした作品をコンスタントに作ればやっていけます。

中小企業がたいした金も使わずに作った駄作を買うより，鳴りもの入りの大資本の作品がよい，という意見もあるでしょうが，ことは資金回収だけにとどまりません。作品がコケたとき，関わっていた人間にも当然しわ寄せがいきます。当然ボーナス減給や，ひどいときはリストラの対象になります。大切な人員の士気や，人員そのものを失う結果が待っているのです。当然ながら，売れない作品を出さないため，メーカーは思いきった冒険をしなくなります。結果として，新規性のある作品の出てくる可能性が減っていき，ゲームの発展は頭打ちになっていくのです。

大手が中小企業レベルの制作内容で作品を作れば……という考え方も当然ありますし，実践している企業もありますが，それでは大手である利点がなにもありません。中小企業のような小回りも利かない大手が，中小企業並みのけちけち政策を取ると，「なにが悲しゅうて大企業で冷や飯食ってなきゃあかんの？」となるのは当然でしょう。そういう場所では業界に失望し，現場を去っていく人も多いようです。

業界が衰退しないためには，中小企業のプロダクション化と大手のディストリビュータ化が必要です。ゲームはいわば，映画とおもちゃの中間にあるとでもいうべきプロダクツです。仕組みの面白さと，そこに仕込まれた世

界観，デザイン，音楽などが渾然一体となってひとつの作品世界を作るのです。

中小企業が人員を集めて作品を作り，大手が支援して，作品を販売する。そうすることで，双方のリスクが減り，思いきった作品を育てていく素地ができますし，中小企業も大企業も，人員を抱え込む必要がなくなり，人員も，さまざまな作品に柔軟に参加していくことができるようになると思うのです。ゲームより歴史の長い映画業界が，これに似た構造をとって成功しているのがなによりの証拠です。

すでに海外ではかなり昔からこういった形態が主流となり，E・Aやブローダーバンドなどの優秀なディストリビュータの下，マイクロ企業や個人が活躍して成功を収めています。アウトソーシングが叫ばれる昨今，日本でもいくつかそういった流れはあるようですが，まだまだ一般的ではありません。業界の真の意味での成熟のために，たとえばCESAのような団体には水掛け論になりがちな再販論争より，こういう点をこそ支援してほしいものです。

最新動向２：技術革新

と，ゲーム業界の近況はこれくらいにして，最新の技術動向にも触れておきましょう。

当連載の技術的要であるDirectXもついにバージョン8になりました。多くの改良がなされましたが，表現能力の各段の向上には驚かされます（バンプ機能なしでも，ガシガシCPUパワーでバンプマッピングしたり，CUBEマップもできますし，ステンシルバッファの強化によって，影の表現などが完璧に近くなりました。HALで動かさないとゲーム向きとはいいがたい速度ですが，ハード未対応でも動くところは本来の意味のDirectXの強みが出てきています）。

ライブラリの使い勝手の進化としては，長いこと別々に進化してきたDirectDrawとDirect3Dの一本化が挙げられるでしょう。かたちとしては，DDとD3Dを分けてとらえることもできますが，一体のインタフェイスとしてとらえたほうが遥かに楽です。ようやく，IM/RMとかいう中途半端に分化した概念から，一連の機能を持つ描画ライブラリとしてとらえられるようになってきました（思えばDirectX7のヘルパーライブラリ強化が，この布石だったといえるでしょう）。まあ，いままでソースを書いてきた身としては，乗り換えの手間で少々文句もいいたいところではありますが，使い勝手の改善ですから，ぐっとこらえることにしましょう（といいつつ，こらえきれずに……以下囲みコラム参照）。

また，このDirectX8がXboxの描画インフラになるのはほぼ確実（X9が出る可能性もありますが，実機登場までの開発スパンからして，DirectX8ベースが主流となるでしょう）ですから，次世代ゲーム機にそのまま移行できる環境が整備されたと思ってまず間違いないでしょう。あと，DirectX8はリッチな人のために（？）マルチCPUにも対応しました。さあ，ガシガシ動かしましょう。

DirectX8の改良点は，表示だけにとどまりません。入力体系，通信，音……それぞれが，激変といってよいほどの進化を遂げているのですが，残念ながら今回の記事ではあまり触れられないので，各自自習してみてください。

また，ハード面でもさまざまな変化が続いています。GeForceの新バージョンがデビューし，次期バージョンも準備段階に入っています。ノートにもハードウェアT＆Lデバイスが登場します。さらに，CPUはついに1GHzの大台を突破しました。

通信インフラも変わりつつあります。フレッツにより，普通の人でも常時接続が楽しめるようになりました。また，NTTのフレッツでのDSL回線や，東京めたりっく通信など，DSL業者の事業が本格化し，ISDNを軽く十倍近く凌駕する500kbit/sec以上の通信線が，これまた普通に扱える値段で登場してきました。今後心配なのはバックボーンの進化です。500kbit/secで10万人がアクセスしたら，あっというまにMAX50Gbit/secのバックボーンが必要になります。いままでは時分割でOKでしたが，今後はストリームなどがより扱われるようになるでしょうから，MAXトラフィックを見据えた回線が必要不可欠です。携帯電話での動画配信を可能にするW-CDMAなどの登場も考えると，きたる2002年にはテラビット級（ギガの1000倍）のバックボーンが必要とされるかもしれません。しかしまあ，そこまで進化すれば，インターネットは電話，テレビを凌駕する通信インフラになるのは確実です。新しい時代の幕開けに期待しましょう。

本題に入る前に：

さて，いよいよ本題に入る前に，まず，前回の記事についていくつか訂正とお詫びを申し上げます。第一に，対応OSをWindows98のみと書いていましたが，Windows2000での動作チェックもできていましたし，VAIOノートでの動作確認（Windows95です）もしていました。 また，一部の環境で正常に動作しないという報告も受けておりますが，この点につきましては，十分なチェック環境を作っていないため，ご迷惑をおかけしております。今回は，より多くの環境でチェックするように努め，皆様に見てもらうようにいたします。

それと，ご報告。Galaxy-Knightsを，本格的に別のメディアで展開していきます。ひとつは，「ハイブリッド携帯」H"上でのコンテンツとして，配信を開始します。もうひとつは現在詳細を詰めていますので，決まり次第またご報告します。期待していてください。では，前置きはこれくらいにして，記事本編に入りましょう。

自分で物体を出したい：3Dデータフォーマット

昔は，3Dソフトを作るうえで問題だったのは，表示速度もさることながら，データを作る環境がないことでした。しかしいまは便利な世の中になり，巷には3Dグラフィックツールと銘打ったソフトが数多く出ています。

当然ながら，ツールにはピンからきりまであり，果ては，SoftImage3D（約150万円）から，無償の「六角大王」まで，さまざまです。そんななかで多くのプロや，アマチュアでも制作する人々に好まれており，広く映像やゲームの制作に使われているのが，LightWave3D（実売約23万円）です。LightWave3Dには，先ほど例として出したSoftImage3Dや，六角大王とのファイルコンバータも発表されているため，データの相互変換フォーマットとしての意味もあり，これひとつで，データ作成（モデリング）から，描画（レンダリング），さらにアニメーション作成と，ムービー作成までこなしてしまう，事実上の業界標準であり，ポータルソフトウェアとしての意味あいもあるといえるでしょう。

このLightWave3Dは，多関節物体のアニメーションを「BONE」と呼ばれる技術でサポートしています。これは，３Dの物体の中に「BONE＝骨」を埋め込んで，骨が関節で曲がるときに，周囲のポリゴンを「肉」のようにまとわりつかせて，骨と一緒に曲げて，アニメーションするという技法です。木構造を持たなければいけないスケルトンと比べて，不定形なもの（たとえば豆腐など）がプルプル動くなどの動きも表現できる大変面白い機能です。

残念ながら，ゲームで使うには「BONE」の技術は処理的に重いので，多くの場合，BONEデータを関節と，関節に影響されるポリゴン群のデータに再編集して使います。ただ，ごく近い将来には，演算がハード化されたGeForceのようなグラフィックカードが主流になるでしょうから，そうなると事情が変わってくるでしょう。

さて，LightWave3Dの物体データは，LWO（LightWaveObject）形式と呼ばれる形式になっています。これは，バイナリデータで，専用ツールでないと見ることができません。今回，データコンバータを用意して，このLightWaveの形式をそのまま使うことを考えていたのですが，時間の都合上割愛します。

その代わりに，LightWave形式をDirectXの標準形式であるX-File（人気テレビシリーズにあらず）に変換するフリーウェアがありますので，こちらを使わせていただきます。LightWave3Dのデータ形式自体は，Newtek（LightWave3D製作元）や，D-storm（国内総代理店）のサイトに詳細が公開されていますので，興味のある方は参考にしてください。

画面に出そう！：描画システム

さて，DirectXでは，3Dデータは「X-File」という形式で取り扱われます。

X-File形式は，当初Direct3D-RMという，描画全般を一通り取り揃えた3Dエンジンのために作られた仕様で，RMを使わない人にはある面「お

まけ」的仕様だったのですが，新しいDirectX8で，これまで複雑に分かれていた描画周りが単一の「DirectX Graphics」というオブジェクトに統一されて，晴れて「DirectXの正式な3Dデータフォーマット」としての地位を確立しました。

囲み記事でも書きましたとおり，第1回でDirectX7の便利なおまけを使っていたら，思いきり裏切られて，四苦八苦してDirectX8に追いつきました。

なんでそこまでして新しいバージョンを追いかけたかといえば，このX-Fileの扱いの変化の恩恵にあずかりたかった，というのが理由のひとつになります。

まあ，そのほか，DirectPlayの機能拡張で，ネットゲームを作りやすくなったという点にも心引かれましたし，さらには2001年秋に登場する，「マイクロソフト初のゲーム専用機＝Xbox」が，DirectX8ベースでの開発体制である，ということもあって，時流に乗り遅れた記事にしたくなかったためでもあります。

そんなこんなで，前回用意したスケルトンプログラムを拡張してみたかったのですが，DirectX8に対応した結果，描画モジュールのいくつかが，似

Microsoftさん，互換性って言葉ご存じですか？ または，DirectX7→DirectX8 地獄の攻防

COLUMN

世の中の進歩に取り残されている間にDirectX8というものが登場してしまいました。「バージョンアップしてよかったじゃないか」そうおっしゃる向きもあると思います。確かにそのとおりです。

同じ描画なのに，なぜか機能単位でバラバラになっていたDirectDrawとDirect3Dがすっきりと統合され，DirectX Graphicsとして生まれ変わりました。また，対応が不十分だったHELに比べ，リファレンスレイヤーというソフトですべて動かしてしまうとんでもない機能を付加したり，さまざまな質的改良があります。DirectPlayによる，通信での音声サポートも嬉しい機能のひとつです。

「じゃあ乗り換えようか？」

……実はここに落とし穴がありました。それも3つも。

1：アンインストールできない？

そうなのです。DirectX8は，一度インストールしたらアンインストールできません（ということになっています）。実際には，アンインストールするためのソフトも出ているようですが，一般的ではありませんし，結構後遺症もきついようです。しかも，DirectXは昔からそうですが，過去のバージョンのDirectXとの共存もできません。これはつまり「DirectX7で作業は続けたいけど，DirectX8性能もちょっと見てみたい」ということができなくなったことを意味します。

2：関数セットが全然違う

DirectX8になって，DirectDrawとDirect3Dが統合されました。それに伴い，DirectDrawは廃止に追い込まれてしまいました（実際には，DirectX7エミュレーションという形式で残されてはいますが，これを使いたい場合，描画のDirectX8への移行はおあずけですし，「バージョンアップしたら遅くなった」という悲惨な結果を招きます）。なにが困るかというと，Bltなど，ビットマップに対する操作がすべて変わってしまったということです。構造体の名前についている番号が7→8とかいうのは，かわいいほうです。同一機能の関数が存在しない場合がままあります。これは結構たまりません。まあでも，ライブラリを直接アクセスしなくても，上位をラップしているヘルパーライブラリでバージョンの違いは吸収できるのでは……。

3：ヘルパーライブラリの嘘つき

と考えたら，大間違いでした。ヘルパーライブラリD3DXは残ってこそいますが，全然別物でした。なぜそうなったかというと，D3DXのやっていたこととほぼ同じ内容をDirectX Graphicsがサポートしてしまったためです。

とはいっても，なんせ肝心要の関数がすっぱりなくなっているわけですから，ダメージは相当のものです。

```
D3DXCreateContextEx(~);
```

デバイスの取得の関数です。これがないと話にならないはずですが，DirectX Graphicsのデバイスの概念ががらっと変わってしまったため，あっさりお払い箱になってしまいました。では，新しい概念ではどうなったのかというと，Direct3D自体の初期化がぐんと簡単になり，

```
lpDirect3D=Direct3DCreate8(D3D_SDK_VERSION);
```

で，Direct3Dオブジェクトが得られ，得られたDirect3Dオブジェクトで，CreateDeviceすれば，Direct3Dデバイスが取得できます。また，このとき面倒だった，「デバイスの列挙」をしなくても，現在のデバイスを調べる命令が付加されたため，初期化は驚くほど簡単です。……でも，チュートリアルを追っかけないとわからないのは大問題。

```
lpD3DX->UpdateFrame(x);
```

画面のフリップ（描画面と表示面の入れ替え）を行うD3DX関数です（でした）。ところが，このフリッピングが，DirectX Graphicsの標準機能に取り込まれたので，あえなく没。新しい概念では，

```
lpDirect3DDevice->Present( NULL,NULL,NULL,NULL )
```

という感じに，Direct3Dデバイス下の制御として単純化されました。

大半の機能においてこんな調子で，「ヘルパーで行わずに，正式ライブラリにしました」のオンパレードになってしまっています。これでは，互換性もへったくれもありません。画面のビットマップ系や，制御系のプログラムは100％書き直しと思って間違いないのが現状です。

ライブラリのアップデートそのものは，なにも悪くないと思います。ただ，DirectXの場合，プログラミングリソースまでごっそり変わってしまっているので問題視しているのです。せっかく作ったプログラム資産が，次のバージョンのライブラリになって，あっさり使えなくなってしまってよいはずがないでしょう。

ライブラリの保守が難しい仕事であるひとつの理由が，「過去への互換性を残す」ことにあるのです。確かに，DirectX8は，バイナリでの互換性は確保されていますが，ライブラリの互換性をここまで切り捨ててしまうと，もはや「上位の製品」とはいえないのではないでしょうか？　これでは同じバイナリリソースを提供する「別物」です。

実験的目的もあるため，この記事ではあえて新しいライブラリに対応しますが，普通こんなことをされたら，最悪の場合プロジェクトが頓挫します（結果として，それを避けるがため，新しいライブラリに移行するのが遅れるわけですね）。

Microsoftの体制に対し，ソフトウェアというものが，積み上げていく資産だということを，再度考えてほしいと痛切に感じました。そういう場合のための「ラップする」ライブラリを提供してくれてもよいと思うのですが，どうでしょうか？（ここで，「昔の組み方がちゃんと提供されているじゃないか」という向きは，再度「バージョンアップ」の意味を考え直してください。なにが悲しくてDirectX8で，LPDIRECTDRAW7オブジェクトを使わなきゃならないのでしょうか）

とまあ，上記のようなわけで，今回のプログラムは，DX7→DX8のインポートの記録でもあります。いちいちどこをどうしたかを挙げているときりがないので書き換えの要点だけ解説します。以下の4点が中心です。そのほかは各自，前回のソースと見比べて自習してください。できれば，Microsoftさんに，正式なDirectX7→DirectX8の資料を配布してもらいたいものです。

1：テクスチャ周り（ああ，作りかけでよかった）

全滅。DirectDrawが実質廃止になり，Direct3Dのテクスチャ操作としての位置づけになりました。これはこれで，体系の簡潔化という意味では非常に意義のあることなのですが，いろいろ整備されていたDirectDrawの機能が，そのまま移行していればなんの問題もないところを，適当にばっさり切り落としたかたちでの移行になっています。たとえば背景画像を出すという簡単な問題でさえ，頭をひねりつつ，チュートリアルやライブラリの機能をしらみつぶしに調べなければならない羽目になっています。

2：D3DXヘルパーによるコンテキスト処理

Direct3D8オブジェクトと，Direct3D8デバイスオブジェクトでの操作への変更。コンテキスト処理はほぼ完全に廃止です。こうなるなら最初からD3DXコンテキストなんか作るな！　これは完全に，DirectX7のヘルパーライブラリの行きすぎが原因ではあるのですが，機能的にあそこまでやったのなら，ちゃんと互換性を確保してくれてもよかったのではないでしょうか。

3：ライブラリ名・マクロ名の変更

まったく同一の機能のものでさえ，なぜか名称の変更が多数行われています。7→8など，バージョン番号の変更は伝統ですからよいとしても，マクロ名の～RENDERSTATE～が～RS～に変わるなど，互換性を潰す変更がかなりあります。互換性を維持した旧来のマクロと，新しいマクロの2つを用意するなどして対処できなかったのでしょうか？　謎です。また，これまでバージョン名がついていなかったライブラリ名に，d3dx8.libのようなバージョン番号が追加されています。

4：頂点，ピクセルのフォーマットの柔軟化

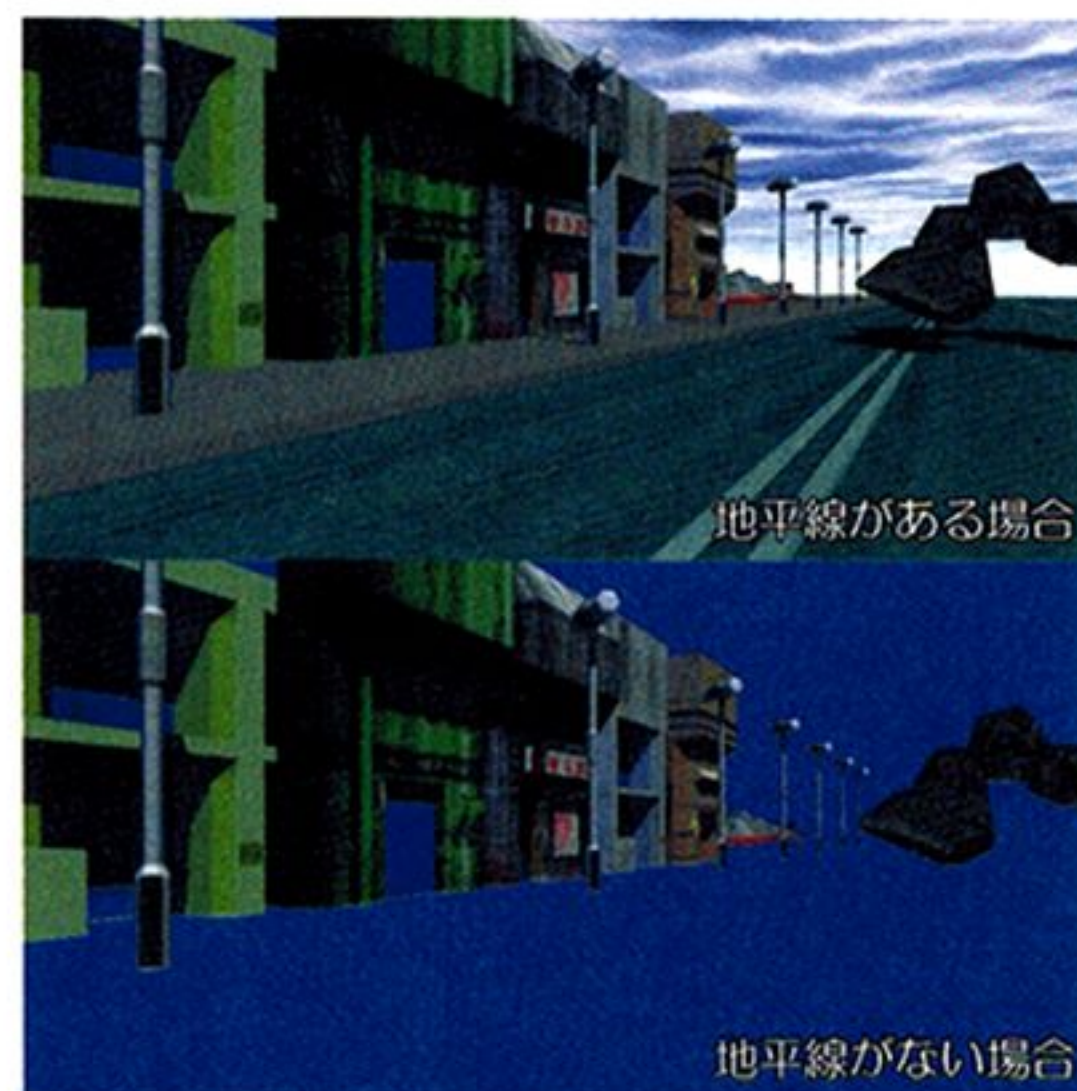
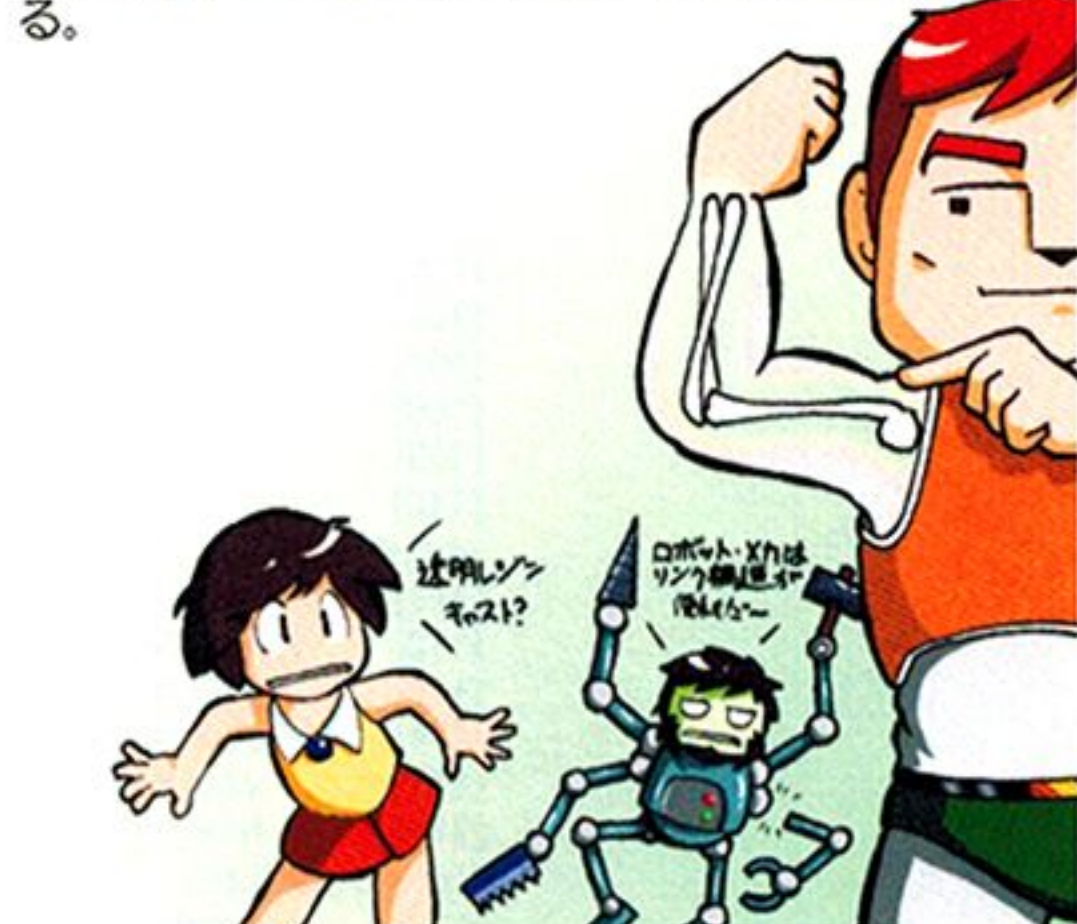
これは機能拡張なのですが，頂点のデータフォーマットや，シェーディングのための手順をユーザーがある程度自由に決められるようになったことでの変更です。いずれにしろ，互換性を保てそうな部分にさえ，非互換の文法があったり（D3DXの物体生成ライブラリは，文法はほぼ同じなのに，引数の順番が入れ替わるという，割ととんでもない非互換があります），いくつか納得できない面があります。いずれにしろ，「非互換」という，ライブラリアップデートとしては通常考えられない障害が問題ですね。

年末のDirect-X8発表で大混乱の図

高いソフトは確かに高機能だが、使いこなせるかどうかは別問題。手軽なものから初めて、ステップアップしていこう。

Boneは周囲のポリゴンを引っ張って変形させることで継ぎ目のない関節を実現する。スケルトン（リンク構造）を外骨格とすれば、Boneは内骨格と言える。

地平線がある場合

地平線がない場合

ても似つかないプログラムになってしまいました。なにが起きたかは囲み記事を参照してください。ここでは，書き換えについては触れず，新しい機能部分について，解説していきます。

DirectX標準の物体データであるX-Fileですが，DirectX8では扱いがとても簡単になりました。ヘルパーライブラリの正式な機能として，X-Fileを扱う関数ができたためです。また，DirectX8と前後して配布された日本語マニュアルには，ちゃんとX-File読み込みプログラムを作るときのチュートリアルまで載っています。さすがにここらへんは，ゲーム機であるXboxを作るための布石なのでしょうか。割としっかりとした土台になったと思えます。

X-Fileも，基本的には市販のツールのデータと同じく，頂点位置の集まり＝ジオメトリ情報と，ポリゴンの属性＝マテリアル情報の集合体です。D3DXライブラリのX-File関数では，

「頂点情報の取得」→「マテリアル数の割り出し」→「マテリアルとテクスチャの読み込み」

という手順でロードを行います。実際の処理は，mapctrl.cppの123～154行で処理を行っていますので参照してください。（サンプルプログラム1）

地面もほしいな：テクスチャと，背景編集

さて，ここまで表示が出ても，なんとなくまともな感じがしないのはなぜでしょうか。前回の記事でも書きましたが，人間は位置判断をする場合，相対判断を基準としています。

「建物が建っている」と考えるには，地平線が必要となるのです。また，地面がないと，移動がなんともメリハリのないものになります。3Dで景観を作る場合，地表は不可欠なものなのです。

図を見ていただければ一目瞭然，その違いがわかると思います。

では，その地表をつけていきましょう。地表を表すのに，もっとも簡単で，もっとも効果的なものは模様＝テクスチャを貼り付けたマス目です。真上から見ると，ちょうど2Dゲームの時代のマップと同じような感じになります。

サンプルプログラム2で，実際に地面をつけてみました。プレイヤーに対応した位置にメッシュを生成し，テクスチャを貼り付けたポリゴンを配置しています。

ただ，それだけだと本当に3Dゲームとなんら変わりないものを，ただ3Dにしているだけになり，少々芸がありません（昔はそれでも十分凄かったのですが）。そこで，マス目＝メッシュをn段とし，それぞれの段の各マス目

表1 Galaxy-Knights MAPDATA－Format (GK_MAPDATA)

ファイル先頭を0としたアドレス	サイズ（バイト）	member name	名称	詳細	内容（固定の場合）
0x00000000	16byte	id_code	ファイル識別子	このファイルが，G.K.のマップデータであることを示します	"""G.K. MAPDATA"",0,0,0,0"
0x00000010	4 bytes	version	バージョン（日付）	ファイルのバージョン。日付管理になっています	0x20001120
0x00000014	4 bytes	objects_top	物体データアーカイブ先頭アドレス	物体データ（X-File形式）が格納されている場所を示します	
0x00000018	4 bytes	positions_top	物体配置データ先頭アドレス	物体の配置データ（前回までプログラム内蔵だったもの）が格納されている場所を示します	
0x0000001C	4 bytes	mesh_top	地表メッシュの先頭アドレス	地表マップ，および高さ情報，イベントなどの設定データが格納されている場所を示します	
0x00000020	4 bytes	tesxture_top	テクスチャデータの先頭アドレス	物体や地表に貼り付けるテクスチャデータ（BMP形式）が格納されている場所を示します	
0x00000024	4 bytes	event_top	イベント名称リストの先頭アドレス	イベント名称リスト（後述）が格納されている場所を示します	
0x00000028	? bytes	mesh_data	地表メッシュデータの実体	メッシュデータヘッダ＋4×Y方向マス目数×X方向マス目数（バイト）の大きさを持つメッシュの実体。フォーマットは後述	
?	? bytes	position_data	物体配置データの実体	物体の配置を記録したデータ	
?	? bytes	object_data	物体データの実体	X-File形式の物体データ列（格納形式は後述）	
?	? bytes	texture_data	テクスチャデータの実体	BMP形式のテクスチャデータ列（格納形式は後述）	
?	? bytes	event_names	イベント名データ列	イベント名称の文字データ列	

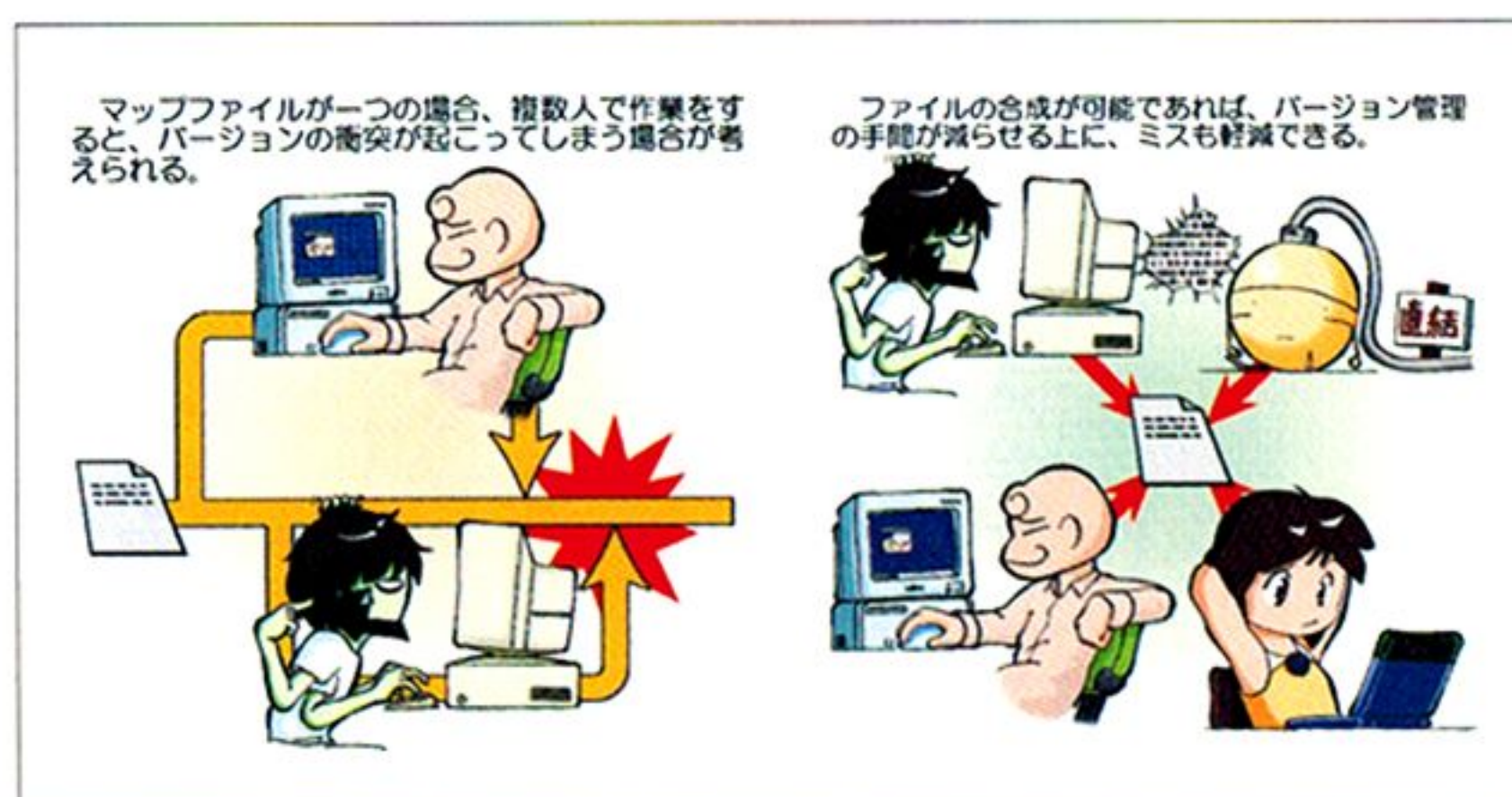
リスト3

```
[1]/*
[2]  Ｏｈ！Ｘ５号
[3]  ＧａｌａｘｙＫｎｉｇｔｓサンプル１
[4]  マップコントロール部分
[5]*/
[6]
[7]#include  "stdafx.h"
[8]#include  "ohx5_1.h"
[9]
[10]// 形状データ（今回のみ内蔵）
[11]long  sdata[]={
[12]  SIMPLE_CUBE,  2000,  4000,  2000,
[13]  FILE_DATA,    1, 150000,
[14]  FILE_DATA,    2, 150000,
[15]  FILE_DATA,    3, 150000,
[16]  FILE_DATA,    4, 150000,
[17]  FILE_DATA,    5, 150000,
[18]  FILE_DATA,    6, 150000,
[19]  FILE_DATA,    7, 150000,
[20]  FILE_DATA,    8, 150000,
[21]  FILE_DATA,    9, 150000,
[22]  SIMPLE_TORUS,  100,  150,
[23]  SHAPE_DONE
[24]};
[25]// マップ（これも今回のみ内蔵）
[26]long  mapdata[]={
[27]  2, -60000,-2000, 60000,  0,0,0,
[28]  3, -60000,-2000, 120000,  0,0,0,
[29]  4,-120000,-2000, 180000,  0,0,0,
[30]  5,-120000,-2000, 120000,  0,0,0,
[31]  6,-120000,-2000, 60000,  0,0,0,
[32]  7, 240000,-2000,-180000,  0,0,0,
[33]  8,-180000,-2000,-180000,  0,0,0,
[34]  9,-180000,-2000,-180000,  0,0,0,
[35]  10,180000,-2000,-120000,  0,0,0,
[36]  10,180000,-2000, -60000,  0,0,0,
[37]  0
[38]};
[39]
[40]// 衝突／イベント判定エリア
[41]// 0＝データ終端
[42]// 1＝衝突判定
[43]// 2＝イベント（イベント言語の起動）
[44]// 3＝ビルトインリンク（配布済みデータの参照）
[45]// 4＝ローカルリンク（同一マップの別空間と接続）
[46]// 5＝メタリンク（同一サーバー別マップと接続）
[47]// 6＝ハイパーリンク（別サーバー別マップと接続）未対応
[48]long  areas[]={
[49]  1,
[50]  3,
[51]  0
[52]};
[53]
[54]char  *file_datanames[]={
[55]  "data¥¥B01_F.x",
[56]  "data¥¥B02_F.x",
[57]  "data¥¥B03_F.x",
[58]  "data¥¥b04_F.x",
[59]  "data¥¥b05_F.x",
[60]  "data¥¥b06_F.x",
[61]  "data¥¥b07_F.x",
[62]  "data¥¥b07_F.x",
[63]  "data¥¥b07_F.x"
[64]};
[65]
[66]void  splitpath( char * fpath,char * fname )
[67]{
[68]char  *fp,*fn,*fpx;
[69]  fp = fpath;
[70]  fn = fname;
[71]  fpx = NULL;
[72]  while( *fn!=0 ){
[73]    if( *fn=='¥¥' ) fpx = fp;
[74]    *fp++ = *fn++;
[75]  }
[76]  *fp = 0;
[77]  if( fpx!=NULL ){
[78]    fpx++;
[79]    *fpx = 0;
[80]  }
[81]}
[82]/*
[83]  形状の初期化
[84]  今回は内蔵データを利用
[85]*/
[86]
[87]void  init_shapes()
[88]{
[89]HRESULT hr;
[90]int sct,name,i;
[91]float  v1,v2,v3;
[92]shapes * spt = shapelist;
[93]char  xfilepath[1024];
[94]char  *xfilename;
[95]  sct = 0;
[96]  max_shapes = 0;
[97]  while( sdata[sct]!=SHAPE_DONE ){
[98]    name = sdata[sct++];
[99]    spt->type = name;
[100]    spt->scale= 1;
[101]    switch( name ){
[102]    case SIMPLE_CUBE:   // 立方体
[103]      v1 = (float)sdata[sct++]/1000; // witdh
[104]      v2 = (float)sdata[sct++]/1000; // height
[105]      v3 = (float)sdata[sct++]/1000; // depth
[106]      hr = D3DXCreateBox( lpD3DD,v1,v2,v3,&(spt->pt),NULL );
[107]      break;
[108]    case SIMPLE_SPHERE:  // 球
[109]      v1 = (float)sdata[sct++]/1000; // radius
[110]      hr = D3DXCreateSphere( lpD3DD,v1,D3DX_DEFAULT,D3DX_DEFAULT, &(spt->pt),NULL );
[111]      break;
[112]    case SIMPLE_CYLINDER:  // シリンダー
[113]      v1 = (float)sdata[sct++]/1000; // base radius
[114]      v2 = (float)sdata[sct++]/1000; // top radius
[115]      v3 = (float)sdata[sct++]/1000; // height
[116]      hr = D3DXCreateCylinder( lpD3DD,v1,v2,v3,D3DX_DEFAULT,D3DX_DEFAULT,&(spt->pt),NULL );
[117]      break;
[118]    case SIMPLE_TORUS:   // トーラス
[119]      v1 = (float)sdata[sct++]/1000; // inner radius
[120]      v2 = (float)sdata[sct++]/1000; // outer radius
[121]      hr = D3DXCreateTorus( lpD3DD,v1,v2,D3DX_DEFAULT,D3DX_DEFAULT,&(spt->pt),NULL );
[122]      break;
[123]    case FILE_DATA:     // X-Flie形式のロード
[124]      LPD3DXBUFFER    xbuf; // ＸＦＩＬＥマテリアルバッファ
[125]      xfilename = file_datanames[ sdata[sct++] ];
[126]      if( xfilename==NULL ) { sct++; break; }
[127]      if( FAILED( D3DXLoadMeshFromX( xfilename,
[128]                D3DXMESH_SYSTEMMEM,
[129]                lpD3DD,
[130]                NULL,
[131]                &xbuf,
[132]                &( spt->mats ),
[133]                &( spt->pt ) ) ) ){
[134]        sct++;
[135]        break;
[136]      }
[137]      D3DXMATERIAL * lpxmats = (D3DXMATERIAL *)xbuf->GetBufferPointer();
[138]
[139]      spt->lpmmats = new D3DMATERIAL8[spt->mats];
[140]      spt->lpmtexs = new LPDIRECT3DTEXTURE8[spt->mats];
[141]
[142]      for( i=0; i<(int)(spt->mats); i++ ){
[143]        spt->lpmmats[i] = lpxmats[i].MatD3D;
[144]        spt->lpmmats[i].Ambient = spt->lpmmats[i].Diffuse;
[145]        // テクスチャを作成する。
[146]        splitpath( xfilepath,xfilename );
[147]        if( lpxmats[i].pTextureFilename != NULL ){
[148]          strcat( xfilepath,lpxmats[i].pTextureFilename );
[149]          if( FAILED( D3DXCreateTextureFromFile( lpD3DD,
[150]                      xfilepath,
[151]                      &( spt->lpmtexs[i] ) ) ) )
[152]            spt->lpmtexs[i] = NULL;
[153]        }
[154]      }
[155]      spt->scale = (float)( sdata[ sct++ ] )/1000;
[156]      xbuf->Release();
[157]      break;
[158]    }
[159]    spt++;
[160]    max_shapes++;
[161]  }
[162]}
[163]
[164]// 形状データ解放
[165]void  release_shapes()
[166]{
[167]DWORD  i,j;
[168]shapes  * spt = shapelist;
[169]  for( i=0;i<max_shapes;i++ ){
[170]    if( spt->type==FILE_DATA ){
[171]      if( spt->lpmmats != NULL ) delete[] spt->lpmmats;
[172]      if( spt->lpmtexs ) {
[173]        for( j = 0; j < spt->mats; j++ ) {
[174]          if( spt->lpmtexs[j] ){ spt->lpmtexs[j]->Release(); }
[175]        }
[176]        delete[] spt->lpmtexs;
[177]      }
[178]    }
[179]    xRelease( spt->pt );
[180]    spt++;
[181]  }
[182]}
[183]
[184]// 物体バッファをひとつ確保
[185]OBJ3D  *make_object( DWORD name )
[186]{
[187]OBJ3D *obj;
[188]  obj = new OBJ3D;
[189]  obj->back = NULL;
[190]  obj->next = objtop;
[191]  if( objtop!=NULL ) objtop->back = obj;
[192]  objtop = obj;
[193]  obj->type = name;
[194]  return obj;
```

```
[195]}
[196]
[197]// 物体バッファをひとつ消去
[198]void    delete_object( OBJ3D *obj )
[199]{
[200]    if( obj==NULL ) return;
[201]    if( objtop == obj ) objtop = objtop->next;
[202]    if( obj->back != NULL ) obj->back->next = obj->next;
[203]    if( obj->next != NULL ) obj->next->back = obj->back;
[204]    xDelete( obj );
[205]}
[206]
[207]// マップから物体リストを初期化
[208]void    init_objects()
[209]{
[210]long    name;
[211]int ct=0;
[212]OBJ3D   ＊obj;
[213]    for(;;){
[214]        name = mapdata[ ct++ ];
[215]        if( name==0 ) break;
[216]        obj = make_object( name-1 );
[217]        obj->pos.x = (float)mapdata[ct++]/1000;
[218]        obj->pos.y = (float)mapdata[ct++]/1000;
[219]        obj->pos.z = (float)mapdata[ct++]/1000;
[220]        obj->rudder.x = D3DXToRadian( mapdata[ct++] );
[221]        obj->rudder.y = D3DXToRadian( mapdata[ct++] );
[222]        obj->rudder.z = D3DXToRadian( mapdata[ct++] );
[223]    }
[224]}
[225]
[226]void    release_objects()
[227]{
[228]OBJ3D   ＊obj = objtop;
[229]OBJ3D   ＊nx;
[230]    while( obj !=NULL ){
[231]        nx = obj->next;
[232]        delete_object( obj );
[233]        obj = nx;
[234]    }
[235]}
[236]
[237]// プレイヤーステータス初期化
[238]void    init_player()
[239]{
[240]    make_myself();
[241]    D3DXVECTOR3 zero;
[242]    zero.x = zero.y = zero.z = 0;
[243]    myself->objp->pos  = myself->objp->rudder =
[244]    myself->objp->move = myself->objp->rotate = zero;
[245]    myself->objp->rudder.y = D3DXToRadian( 45 );
[246]}
[247]// プレイヤーによる制御
[248]void    player_drive()
[249]{
[250]WORD    key = GetKeys();
[251]    command_player( myself->objp,( key << 16 )| dpmsg_data_packet );
[252]    send_player_control( myself->dplayID, key );
[253]
[254]    camera_pos = myself->objp->pos;
[255]    camera_rud = myself->objp->rudder;
[256]}
[257]
[258]
[259]// プレイヤーの挙動
[260]// 複数プレイヤー対応版
[261]void    command_player( OBJ3D ＊ob,DWORD cmd )
[262]{
[263]WORD    key = (WORD)( cmd>>16 );
[264]
[265]    if( key & PAD_CMD ){
[266]// 左右に平行移動
[267]// 上下に平行移動
[268]
[269]    } else {
[270]// 左右に旋回
[271]        if( key & PAD_LEFT ){
[272]            if( ob->rotate.y<0.05f ) ob->rotate.y += 0.002f;
[273]        } else if( key & PAD_RIGHT ){
[274]            if( ob->rotate.y>-0.05f ) ob->rotate.y -= 0.002f;
[275]        } else {
[276]            if( ob->rotate.y>0.002f ) ob->rotate.y ＊= 0.8f; else ob->rotate.y = 0;
[277]        }
[278]// 上下に旋回（最大上下30度。旋回していないときは，自動的に水平に戻る）
[279]        if( key & PAD_UP ){
[280]            if( ob->rotate.x<0.05f ) ob->rotate.x += 0.002f;
[281]        } else if( key & PAD_DOWN ){
[282]            if( ob->rotate.x>-0.05f ) ob->rotate.x -= 0.002f;
[283]        } else {
[284]            ob->rotate.x = 0;
[285]        }
[286]    }
[287]// 前進
[288]    if( key & PAD_FORWARD ){
[289]        if( ob->move.z<1 ){
[290]            ob->move.z += 0.002f;
[291]        } else {
[292]            ob->move.z = 1;
[293]        }
[294]    } else {
[295]        if( ob->move.z>0.002f ) ob->move.z ＊= 0.8f; else ob->move.z = 0;
[296]    }
[297]// ベクトル加算
[298]    ob->rudder.y += ob->rotate.y;
[299]    if( ob->rotate.x!=0 ){
[300]        ob->rudder.x += ob->rotate.x;
[301]        if( ob->rotate.x<0 ){
[302]            if( ob->rudder.x<-0.7f ) ob->rudder.x = -0.7f;
[303]        } else {
[304]            if( ob->rudder.x> 0.7f ) ob->rudder.x = 0.7f;
[305]        }
[306]    } else {
[307]        ob->rudder.x ＊= 0.8f;
[308]        if( ob->rudder.x<0.01f && ob->rudder.x>-0.01f ){ ob->rudder.x = 0; }
[309]    }
[310]// 移動ベクトル生成
[311]    D3DXMATRIX  mat;
[312]    D3DXMatrixRotationY( &mat,-ob->rudder.y );
[313]    D3DXVECTOR3 mv;
[314]    mv = ob->move;
[315]    D3DXVec3TransformNormal( &mv,&mv,&mat );
[316]    ob->pos += mv;
[317]}
```

に高さの情報を持たせることにします。こうすることで，3次元方向に広がるマス目を簡単に描くことができるようになります。
(サンプルプログラム3：このサンプルは，インターネットからのダウンロードとなります)

建物，物体，移動体，地形：統合マップデータ

地表ができて，その上に建物を建てることができ，どうにか街の景観を作れるようになってきました。しかし，データをプログラムに埋め込んでいたり，かなり不恰好です。のちのちのことを考えると，マップデータはマップデータでまとまっていてほしいですね。そこで，マップに関するデータを1個のフォーマットに整理することにします。

フォーマットを1個にしてしまうと，いくつかの利点と欠点が出てきます。

利点：ファイルの数が少なくなるので，データの管理が簡単になります。また，データ読み込みの時間も短縮されます
欠点：共同作業の場合，1個のファイルを複数の人間がいじる危険性が出ます（バージョンの混乱が起きる可能性があります）

利点のほうはたいへんありがたいものですが，欠点は，ひとつ間違うとバグの原因になりかねません。特にゲームはこの規模になると共同作業でないとなかなか作ることができませんので，なんとかしたい欠点です。

そこで，あとから内容をつなぎあわせることのできるフォーマットを考えました。具体的にどうするかというと，データの各要素をブロックに分けて，ブロックを繋ぐフォーマットを規定し，あとから自由に統合，分離できるようにしようというものです。

既知のデータであるビットマップファイル（BMP）と，X-File（x）についてはファイル名も含めてデータに格納し，それ以外は親ファイルに別の拡張子をつけて分離できるようにします。

では，以下にそのフォーマットを規定します。なお，データは下位バイトがアドレスの小さいほうにくる形式（リトルエンディアン）を採用します。

先頭のファイル識別子により，このデータが正式なG.K.用のデータであるかどうかを判断できます。また，2番目のバージョンにより，新しいデータバージョンのファイルを作る必要が出ても，旧データとの間で互換性を保つことができます。

3番目以降の項目は，実データの格納状況を示すものです。実際は，mesh_data以後のデータの位置は順不同で，さらに間にどんなデータが入っていても構いません。なぜかというと，参照はそれ以前に格納されたポインタを使い，たとえばメッシュデータの場合，ファイルがmapdata以降にロードされているとすると，

```
MESH_DATA * mesh_address;
mesh_address = (MESH_DATA * ) ( (long)( &mapdata )+
mapdata->mesh_top );
```

という風に求められるからです。各実体側に，実体の実際のサイズや要素数のデータを含めるので，それぞれのデータの実体も独立しています。こういうデータ構造だと，あとでデータに追加項目がほしい場合，簡単に増設できるのでたいへん便利です。

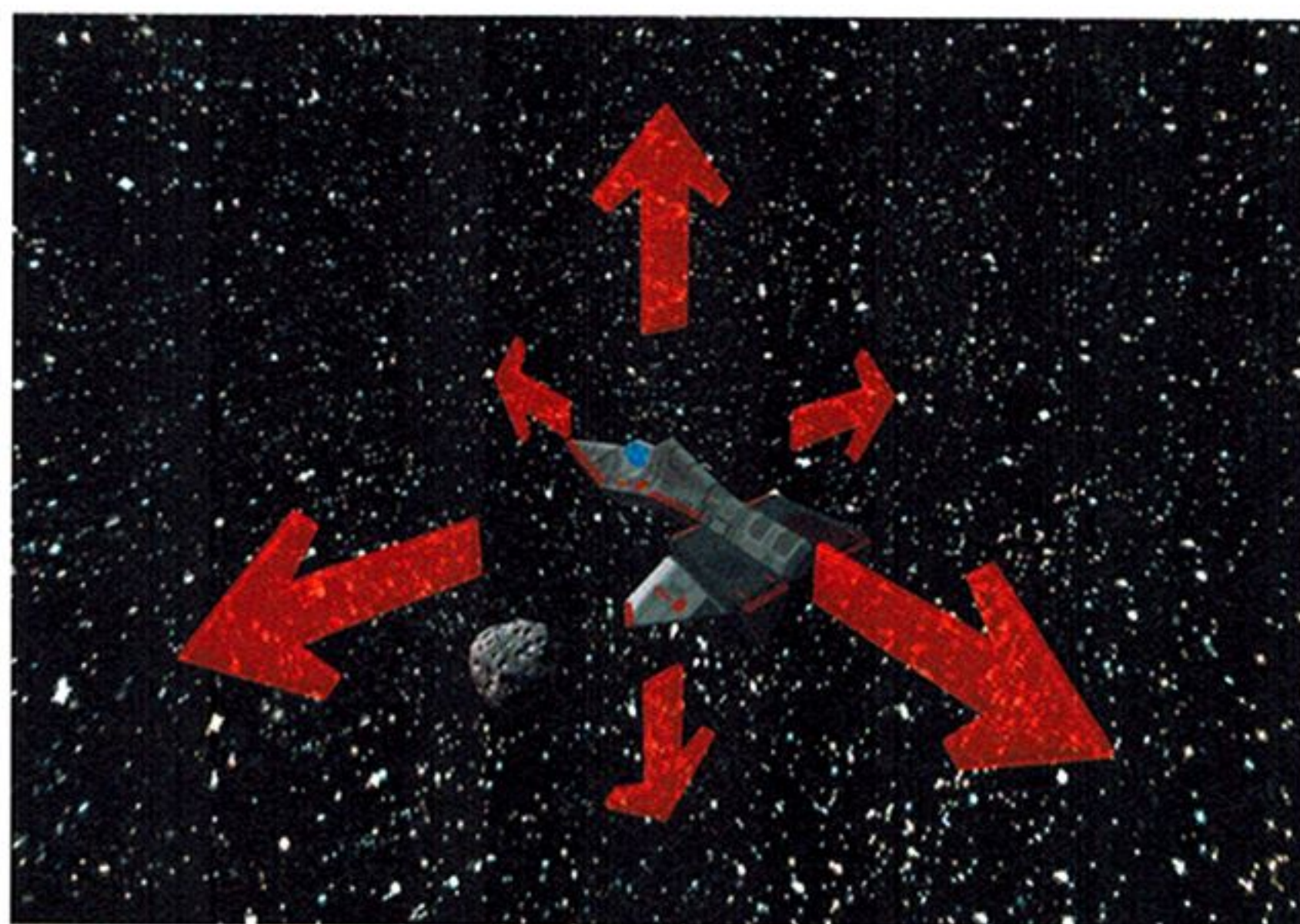
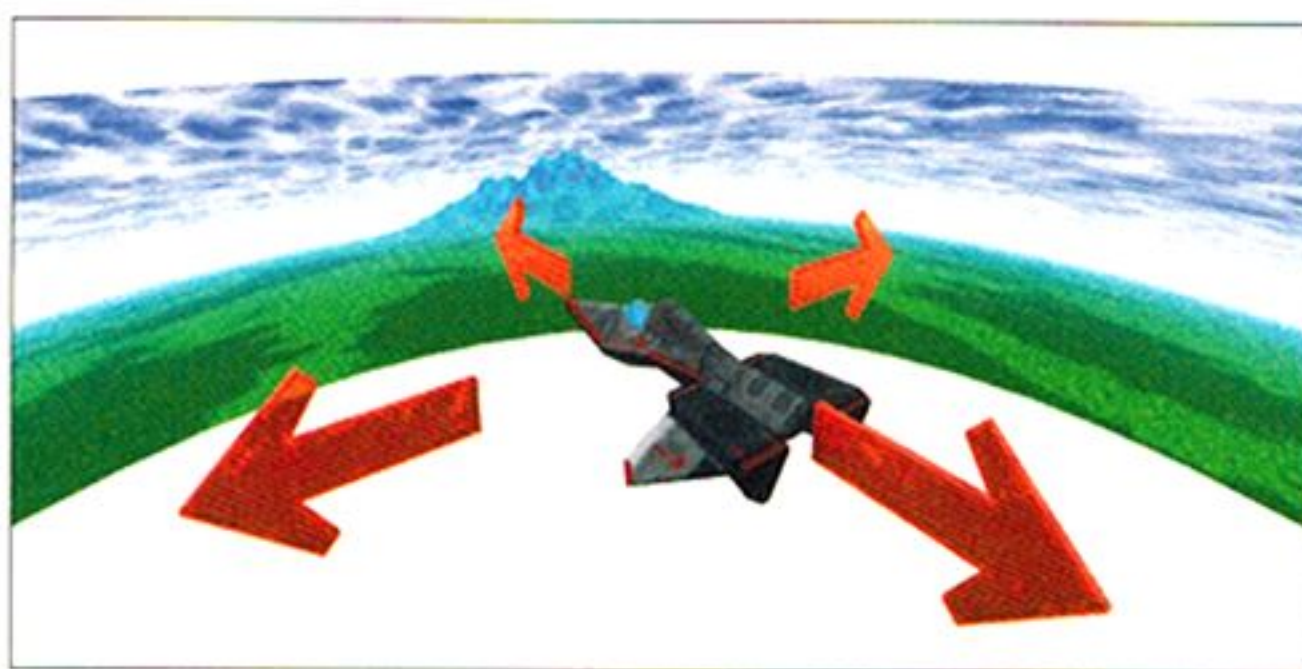
さて，各実体のデータは以下のようになっています。本体と同じくデータはすべてアドレスの小さいほうに下位バイトがくる「リトルエンディアン」です。なお，このフォーマットに対応したバージョンのソースプログラムは，弊社サイバーヘッドのホームページからのダウンロードで提供いたします。

地面に縛られるのは嫌だ！：空中(宇宙)マップと移動制御系

地上の景観はできましたが，それだけではGalaxy-Knightsの「Galaxy」が泣きますね。やはり宇宙を出さずしては話が進みません。では，宇宙を出してみましょう。ことは簡単です。

マップシステムは3Dのマス目になっていますので，高さ方向にマップを構成するのは簡単です。あとは，地面をなくして，かつ，自分の姿勢制御を3軸の回転にするだけでよいのです。

宇宙の場合，物体の配置単位（＝マス目）はかなり大きくしたほうがリアルになります。地上ではひとつのマス目を1歩分の移動間隔とほぼ等価と考えましたが，宇宙の場合は「制御区画」くらいの考えにしたほうがなにかと都合がよいのです。データも大きくなりませんし，なにより，配置したものの位置感覚がわかりやすくなります。

実際の対応ソースは，インターネットからのダウンロードとさせていただきます。

移動が三次元になると，移動や方向転換が難しくなる。その結果，戦闘や索敵でのプレイヤーの負担が大きくなってしまう

移動が二次元の場合，表示されるオブジェクトが立体物でも，移動や索敵は平面上に制限される為に簡単な操作で済む。プレーヤーの負担も少ない

お詫び：
今回，サンプルプログラムおよびツールの収録が間にあわなかった部分については弊社サーバからのダウンロードとさせていただきました。サポート情報はhttp://www.cyberhead.co.jp/を，実際のソースやコンテンツなどに関してはhttp://galaxyknights.com/をそれぞれ参照してください。

表2　地表メッシュデータの実体 (GK_MESH_DATA)

データブロックの先頭を0としたアドレス	サイズ(バイト)	member_name	名称	詳細
0x00000000	4 bytes	sizex	メッシュデータのX方向の個数	メッシュの横幅
0x00000004	4 bytes	sizey	メッシュデータのY方向の個数	メッシュの縦幅
0x00000008	4 bytes	sizez	メッシュデータの縦方向の枚数	メッシュの重なった枚数
0x0000000C	4 bytes	cell_size	データセル(メッシュの1区画)のデータサイズ	デフォルトは4
0x00000010	4 bytes	spanx	X方向のメッシュの空間サイズ	空間でのメッシュ1個単位の大きさを示す
0x00000014	4 bytes	spany	Y方向のメッシュの空間サイズ	同上
0x00000018	4 bytes	spanz	Z方向のメッシュの空間サイズ	同上
0x0000001C	4 bytes	attributes	データ属性	格納データの所属性が格納されている bit0 = X lap around bit1 = Y lap around
0x00000020	sizex_sizey_sizez_cell_size bytes	body	データ本体	セルが下記の順序で格納されている

"(x0,y0,z0)"	"(x1,y0,z0)"	....	"(xn,y0,z0)"
"(x0,y1,z0)"	....		
			"(xn,yn,z0)"
	....		"(x0,y0,z1)"
	"(xn,yn,z1)"		
	....	....	"(xn,yn,zn)"

セル（メッシュデータ）1個の標準フォーマット（bodyに格納されているデータひとつの内容）

b31-b27	b26-b24	b23	b22	b21b	20-b16	b15-b8	b7-b0
予備	テクスチャサイズ $2^{(1+n)}$になる 例：n=3 なら，メッシュのサイズは16ドット	メッシュの存在フラグ (=0でこのメッシュは透明)	縦反転	横反転	テクスチャページ (格納データの何番目を使うか)	データの開始場所Y	データの開始場所x

なお，上記は独立したデータとして分離セーブ，およびファイルとの再統合が可能とします

表3

物体配置データの実体(GK_POSITION_DATA)

データブロックの先頭を0としたアドレス	サイズ（バイト）	member_name	名称	詳細
0x00000000	4 bytes	count	座標データ個数	格納されている座標データの総数
0x00000004	4 bytes	cell_size	各座標データのサイズ	格納されている座標データの1個のサイズ(標準は40)
0x00000008	count * cell_size bytes	body	データ本体	データ本体が格納されている

配置データ1個の標準フォーマット(bodyに格納されているデータひとつの内容)

データブロックの先頭を0としたアドレス	サイズ（バイト）	member_name	名称	詳細
0x0000	4 bytes	type	格納タイプ	タイプ値 / 意味 SIMPLE_CUBE / 直方体 SIMPLE_SPHERE / 球体 SIMPLE_CYLINDER / 円筒 SIMPLE_TORUS / ドーナツ FILE_DATA / X-File形式
0x0004	4 bytes	posx	中心座標X成分	
0x0008	4 bytes	posy	中心座標Y成分	
0x000C	4 bytes	posz	中心座標Z成分	
0x0010	4 bytes	sizex	物体のX方向拡大率	
0x0014	4 bytes	sizey	物体のY方向拡大率	
0x0018	4 bytes	sizez	物体のZ方向拡大率	
0x001C	4 bytes	attr	物体アトリビュート	単純図形の場合色成分，物体の場合X-File通し番号
0x001C	2 bytes	rudx	X軸方向の回転	4096で1回転
0x001C	2 bytes	rudy	Y軸方向の回転	同上
0x001C	2 bytes	rudz	Z軸方向の回転	同上
0x001C	2 bytes	pad	ワード境界補正ダミー	構造体が4で割り切れるバイト数にするためのダミー値

表4　物体データの実体(GK_OBJECT_DATA)

データ1個先頭を0としたアドレス	サイズ（バイト）	member_name	名称	詳細
0x0000	4 bytes	count	物体データ個数	X-Fileの個数
0x0004	count * 4 bytes	table	X-File格納アドレステーブル	各X-File先頭アドレスに対するデータブロック先頭からのオフセット
0x0008	? byte	body	データ実体	"ファイル名の長さ(1byte),ファイル名文字列,0,Xファイル実体...."の形式でデータが格納されている。

表5　テクスチャデータの実体(GK_TEXTURE_DATA)

データブロックの先頭を0としたアドレス	サイズ(バイト)	member_name	名称	詳細
0x00000000	4 bytes	count	テクスチャデータ個数	格納したBMPファイルの総数
0x00000004	count * 4 bytes	table	テクスチャ格納アドレステーブル	各BMPファイル先頭アドレスに対するデータブロック先頭からのオフセット
0x00000008	? bytes	body	テクスチャ実体	"ファイル名の長さ(1byte),ファイル名,0,BMPファイル実体...."の形式でデータが格納されている

表6　イベント名称データ列の実体(GK_EVENT_NAMES)

データブロックの先頭を0としたアドレス	サイズ(バイト)	member_name	名称	詳細
0x00000000	4 bytes	count	イベント総数	格納したイベント名称の総数
0x00000004	count * 2 bytes	table	イベント格納アドレステーブル	各文字列先頭アドレスに対するデータ先頭からのオフセット(の，下位2バイト)
0x00000008	? bytes	strings	文字列実体	"文字列の長さ(1byte),文字列,0 ....."の形式でデータが格納されている

実際のツール，および対応ソースは，インターネットからのダウンロードで提供いたします。参考にしてください。

パズルでプログラミング　第1回
バックトラックと再帰の基本

広井 誠　Hiroi Makoto

プログラミングでアルゴリズムを学ぶときによい題材となるのがパズルの解法です。優れたアルゴリズムを組み合わせることで，より効率よく巧みにパズルを攻略することができます。ここではC言語を用いてパズルの解法プログラミングと各種アルゴリズムを追っていきます。第1回はバックトラックです。

はじめに

最近Windows上で動作し，フリーで利用できる開発環境が増えてきています。PerlやTcl/Tkといった海外で開発されたスクリプト言語だけではなく，日本で生まれたスクリプト言語であるRubyやHSPなどが注目を集めています。Rubyは，まつもとひろゆき氏が開発したオブジェクト指向スクリプト言語で，HSP（Hot Soup Processor）はonion software（onitama氏）が開発したBASICに似たスクリプト言語です。このようにフリーで利用できる開発環境が増えると，自分の好みにあった言語を選ぶことができるので，昔に比べるとWindows上でも気軽にプログラミングを楽しめる環境が整ってきているように思います。

さて，プログラミングを楽しむにしても，それではなにを作ろうかと悩む方もいるでしょう。このような人にぴったりの題材が「パズルの解法」です。パズルを解くという明確な目標があり，そしてなによりも実際にパズルを解いたときの喜びは大きく，プログラムを作る意欲をかき立ててくれます。では，どうやったらパズルを解くことができるのでしょう。また，パズルに限らず，教科書の例題は理解できても，そこから一歩でもはずれると，どうやってプログラムを作ったらよいのかわからない，という方もいると思います。

プログラミングの上達の秘訣は，実際にプログラムを作って動作を確認することです。このとき，プログラミング言語の文法を覚えることも必要ですが，それだけでは簡単なプログラムしか作ることができません。特に，最近の優秀なツールを使えば，部品を配置するだけで簡単にインタフェイスを作ることができます。ところが，見栄えがよくても中身がしっかりしていないと満足のいくアプリケーションにはなりません。

ここで重要なのが「アルゴリズム」です。アルゴリズムとは，ある問題を解く手順のことを意味し，この手順を特定のプログラミング言語で記述したものが，実際のプログラムとなります。プログラミング言語の文法を理解したとしても，アルゴリズムがわからないのではプログラムを作ることはできませんね。そして，もうひとつ重要なのが「データ構造」です。データ構造とは，データの表現方法のことですが，アルゴリズムがわかっていてもデータ構造の選択が間違っていると，処理に時間がかかりすぎて現実的な時間で答えを求めることができないということもあるのです。

特にパズルを解く場合，アルゴリズムとデータ構造の選択は重要です。パズルによっては，初めから巧妙な解法を思いつく人もいるのですが，筆者のような凡人に，天才的なひらめきを期待するのは無理というものです。しかしながら，基本的なアルゴリズムとデータ構造を使うことで，ベストではありませんが，そこそこのプログラムを作ることは可能です。

コンピュータ科学の歴史は半世紀ちょっとしかありませんが，いままでに数多くの優れたアルゴリズムやデータ構造が考案されています。たとえば，データの整列（sort）にはクイックソートやマージソートなど，データの探索（search）にはハッシュ法や二分探索木などがあり，ほかの分野にもさまざまな優れたアルゴリズムが知られています。これらをうまく使いこなすことができれば，あなたのプログラミングスキルもレベルアップすることは間違いありません。そして，このようなアルゴリズムを理解するのに適しているのが，パズルの解法なのです。つまり，

パズルはプログラミングの学習に最適！

というわけです。使用するプログラミング言語はC言語としますが，アルゴリズムやデータ構造は，プログラミング言語に依存するものではありません。ほかの言語へ移植するのもプログラムの勉強になるでしょう。

さて，前置きが長くなりましたが，そろそろ本題に入りましょう。今回はパズルの解法で基本となるアルゴリズム「バックトラック」を説明します。

再帰呼び出し

ところで，アルゴリズムとかパズルの解法というと，避けては通れないのが「再帰呼び出し」です。再帰呼び出しとは，関数定義のなかで自分自身を呼び出すことです。簡単な例題として，階乗を計算する関数を考えてみましょう。階乗は次のように定義することができます。

階乗の定義
0! = 1
n! = n ＊ （n - 1）!

n!の定義に（n-1）!が使われていますね。このように，階乗は自分自身を使って再帰的に定義されています。つまり，n!を求めるためには（n-1）!を計算すればいいわけです。そして，（n-1）!を求めるためには（n-2）!を計算し，（n-2）!を求めるため（n-3）!を計算する，というように最後には0!を求めることになります。これは階乗の定義から1であることがわかります。結局n!を求めるには，1からnまでの整数を乗算すればいいのですが，再帰呼び出しを使うと階乗の定義そのままにプログラムすることができます。

```
/＊ 階乗の計算 ＊/
int fact（int n）
{
  if（n == 0）{
    return 1;
  } else {
    return n ＊ fact（n - 1）;
  }
}
```

関数factは引数nが0であれば1を返し，そうでなければn＊fact（n-1）の計算結果を返します。factの定義でfact自身を呼び出していますね。これ

が再帰呼び出しです。

関数定義のなかで自分自身を呼び出すことができるなんて，なにか特別な仕掛けがあるのではないか，と思ってしまうかもしれませんね。筆者も最初に再帰呼び出しを見たときは，関数を定義するのに自分自身を呼び出すなんて，ヘビが自分の尻尾を食べていくような奇妙な感覚に思えて，なかなか理解できませんでした。ところが，再帰呼び出しは特別なことではなく，近代的なプログラミング言語であれば，ほぼどれも再帰呼び出しを使うことができるのです。

階乗と同じように再帰定義で表されるアルゴリズムはたくさんあります。階乗の計算は簡単なので，再帰呼び出しを使わなくても繰り返しでプログラムすることができますが，再帰で定義されるアルゴリズムのなかには，繰り返しに変換するとプログラムが複雑になってしまうものがあります。このような場合は，素直に再帰呼び出しを使ってプログラミングしたほうがわかりやすいプログラムとなり，間違いを犯す（バグを生み出す）危険性が少なくなります。難しいアルゴリズムでも，再帰呼び出しを使うと簡単にプログラムできる場合もあるのです。

一般に，再帰呼び出しは難しいテクニックと思われているようで，初心者の方は避けて通ることが多いように思います。ところがLispというプログラミング言語では，再帰呼び出しは初心者が覚えるべき基本テクニックのひとつにすぎません。慣れるまでちょっと苦労するかもしれませんが，基本を理解すれば簡単に使いこなすことができるようになります。プログラミングに興味をお持ちの方は，ぜひ再帰呼び出しをマスターしてください。

それでは，再帰呼び出しのポイントを説明しましょう。図１を見てください。

Call:1 →	Call:2 →	Call:3 →	Call:4 →	Call:5
n:4	n:3	n:2	n:1	n:0
value : 24 ←	value : 6 ←	value : 2 ←	value : 1 ←	value : 1

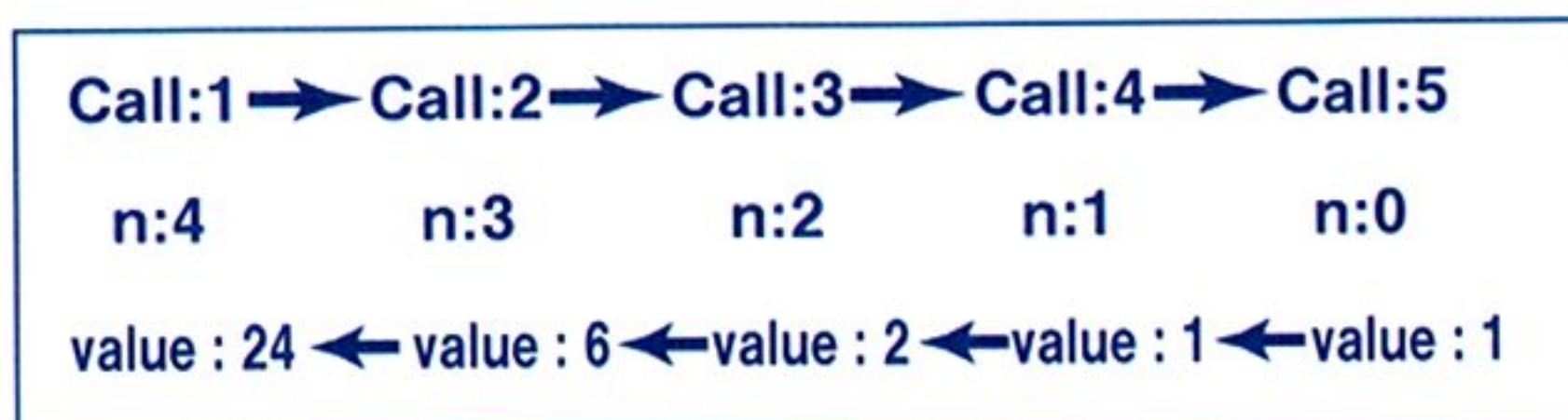



図１　fact の再帰呼び出し（n：引数の値，value:返り値）

図１は関数fact (4) の呼び出しを表したものです。最初の呼び出し（Call:1）では，引数nの値は4なのでnの値を1減らしてfactを再帰呼び出しします。２回目の呼び出しでは，引数nの値に3が代入されます。ここで，最初に呼び出したときと，２回目に呼び出したときでは，引数nの値が違うことに注意してください。

関数の引数は局所変数として扱われます。局所変数には有効範囲（スコープ）があります。引数の場合，その関数が実行されているあいだだけ有効です。局所変数は，関数呼び出しが行われるたびに新しいメモリに割り当てられ，そこに値が代入されます。そして，関数の実行が終了すると，局所変数用に割り当てられたメモリは解放されます。つまり，１回目の呼び出しと２回目の呼び出しでは，引数nに割り当てられるメモリが異なっているのです。ここが再帰呼び出しを理解するポイントのひとつです。

●ポイント１

関数呼び出しが行われると，局所変数は新しいメモリに割り当てられる。

プログラムを見ると変数nはひとつしかありませんが，再帰呼び出しが行われるたびに新しい変数nが作られていくと考えてください。fact (4) を実行しているときのnは4であり，fact (3) を呼び出すときには，このnの値を書き換えるのではなく，新しい変数nを用意して，そこに3を代入するのです。現在のプログラミング言語では，局所変数はあって当然の機能です。再帰呼び出しを使いこなすためにも，局所変数の有効範囲はきちんと理解しておきましょう。

同様に再帰呼び出しが行われ，５回目の呼び出し（Call:5）で引数nが0になります。このとき，ifのthen節が実行され1が返されます。ここで再帰呼び出しが止まります。ここが第２のポイントです。

●ポイント２

再帰呼び出しを止める条件（停止条件）を設定すること。

停止条件がなかったり，あってもその条件を満たさない場合，関数を際限なく呼び出すことになり，Ｃ言語であればプログラムは暴走することになります。再帰呼び出しを使う場合は，この停止条件に十分注意してください。慣れないうちは図を描いてみるのもいいでしょう。

fact (0) は1を返してfact (1) に戻ります。fact (1) を実行しているあいだ，引数nの値は1ですね。したがって，fact (1) の返り値は1＊1を計算して1となります。あとは同様に，再帰呼び出しした関数の返り値を使って計算し，最後にfact (4) の値24を求めることができるのです。関数factは自分自身を１回だけ呼び出しています。これを一重再帰と呼びます。このような再帰呼び出しは，繰り返しへ簡単に変換することができます。

もうひとつ簡単な数値計算の例を示しましょう。フィボナッチ関数も再帰的に定義される関数です。

フィボナッチ関数の定義

$$f(n) = \begin{cases} 1; & n = 0 \\ 1; & n = 1 \\ f(n-1) + f(n-2); & n > 1 \end{cases}$$

1, 1, 2, 3, 5, 8, 13 .... という直前の２項を足していく数列

フィボナッチ関数も再帰呼び出しを使えば簡単にプログラムできます。

```
/* フィボナッチ関数 */
int fibonacci (int n)
{
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci (n - 1) + fibonacci (n - 2);
  }
}
```

fibonacciはfactとは違い，自分自身を２回呼び出しています。このことを二重再帰といいます。fibonacciの呼び出しをトレースすると図２のようになります。

```
f (5)─┬─f (4)─┬─f (3)─┬─f (2)─┬─f (1)
      │       │       │       └─f (0)
      │       │       └─f (1)
      │       └─f (2)─┬─f (1)
      │               └─f (0)
      └─f (3)─┬─f (2)───f (1)
              │   └─f (0)
              └─f (1)
```

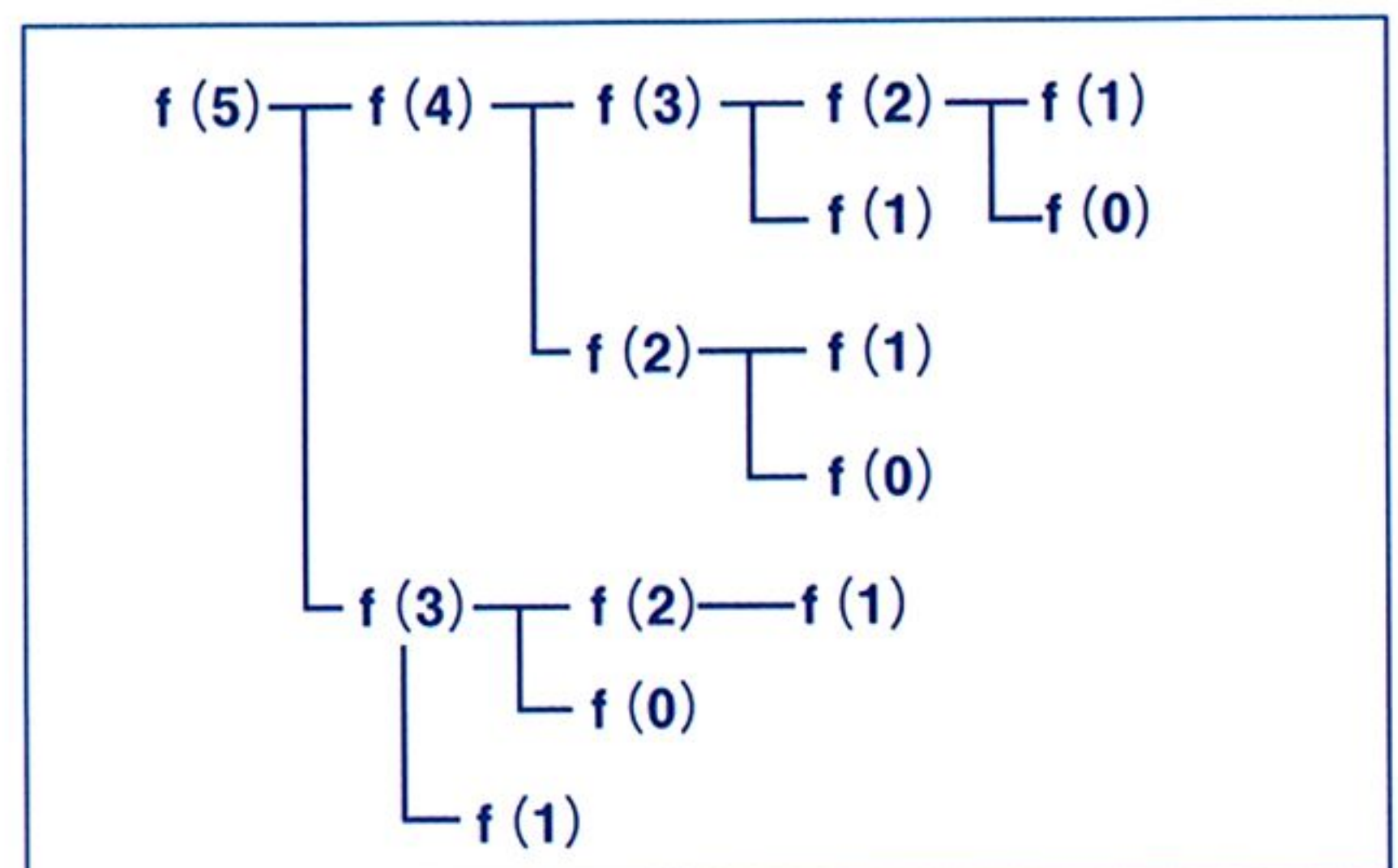



図２　fibonacci のトレース

同じ値を何回も求めているため，効率はとても悪いのです。この処理は配列を使うと高速化することができます。何度も同じ計算をしないように，計算した値は配列に格納しておいて，２回目以降は配列から計算結果を求めるようにします。また，フィボナッチ関数の値は，整数（int）の範囲では高々45までしか求めることができません。あらかじめ0から45までの値を計算して配列に格納しておけば，もっと簡単に値を求めることができます。

このような方法を「表引き」とか「表計算法」といいます。アルゴリズムによっては，表計算法を使うことで処理を劇的に高速化することができます。もちろん，パズルの解法にも応用することができますが，今回の主題である再帰呼び出しとバックトラックから脱線するので，この辺で話を打ち切ります。今後の楽しみにとっておきましょう。

経路の探索

次は「バックトラック（backtrack）」というアルゴリズムを説明します。バックトラックは，パズルの解法でよく使われる，王道ともいうべきアルゴリズムです。たとえば，迷路を考えてみましょう。ある地点Aで道が左右に分かれていたとします。左の道を選んで先へ進むと，行き止まりになってしまいました。この場合，A地点まで戻り右の道へ進まなければいけません。このように，失敗したら後戻りして別の選択肢を選び直す，という試行錯誤を繰り返してゴールにたどり着く方法がバックトラックなのです。バックトラックはパズルの解法だけではなく，いろいろな分野の問題に応用できるアルゴリズムです。そして，再帰呼び出しを使うと簡単にプログラムすることができます。

それでは，図3に示す簡単な経路図を使って，具体的にバックトラックを説明します。

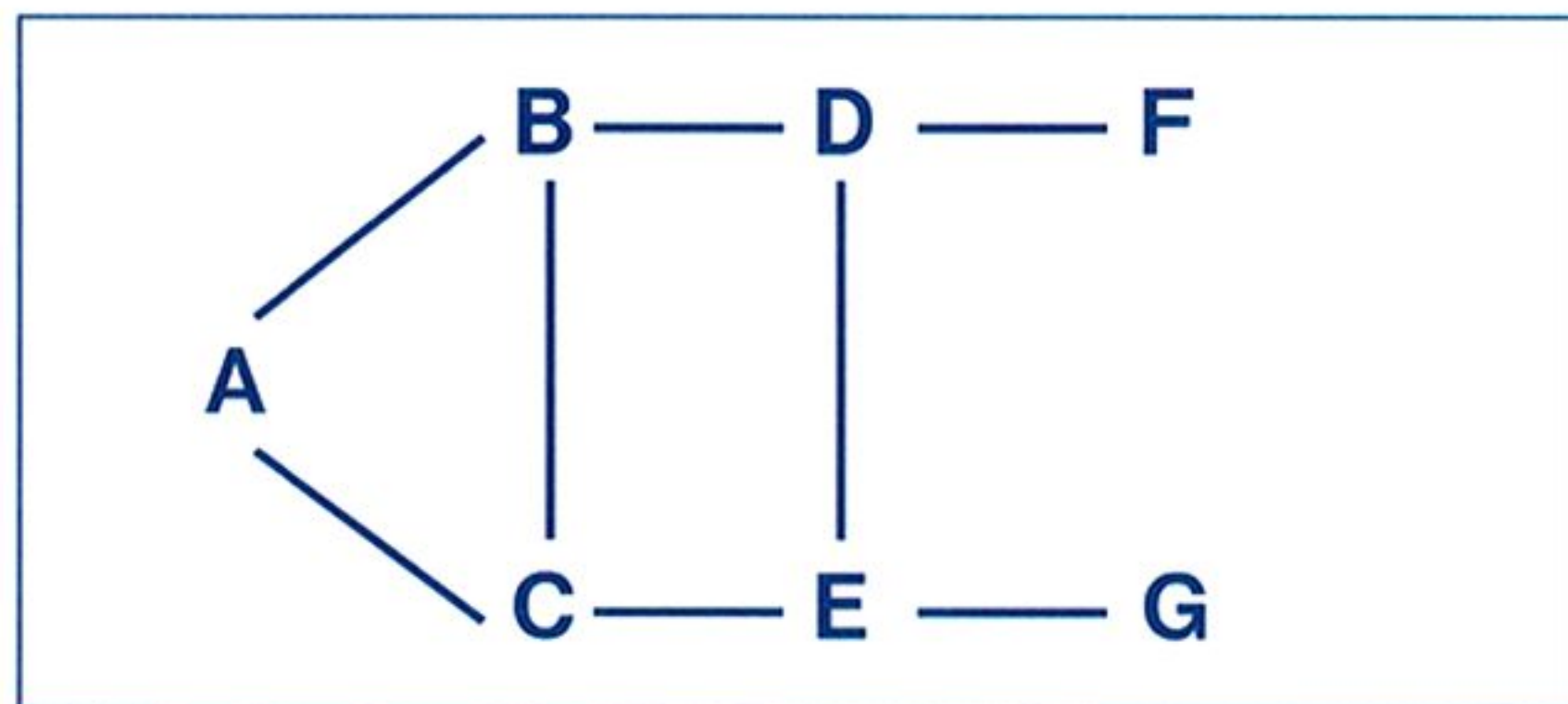



図3　経路図

点とそれを接続する線からなる図形を「グラフ（graph）」といいます。点のことを「頂点（vertex）」とか「節（node）」と呼び，線のことを「辺（edge）」とか「弧（arc）」と呼びます。グラフには2種類あって，辺に向きがないものを「無向グラフ」といい，向きがあるものを「有向グラフ」といいます。有向グラフは一方通行の道と考えるとわかりやすいでしょう。図3ではアルファベットで頂点を表しています。今回は経路をグラフで表していますが，このほかにもいろいろな問題をグラフで表現することができます。

グラフをプログラムする場合，よく使われる方法が「隣接行列」と「隣接リスト」です。隣接行列は2次元配列で頂点の連結を表す方法です。頂点がN個ある場合，隣接行列はN行N列の行列で表すことができます。図3を隣接行列で表すと，図4のようになります。

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	1	1	0	0	0
C	1	1	0	0	1	0	0
D	0	1	0	0	1	1	0
E	0	0	1	1	0	0	1
F	0	0	0	1	0	0	0
G	0	0	0	0	1	0	0

図4　隣接行列

Aに接続している頂点はBとCなので，A行のBとCに1をセットし，接続していない頂点には0をセットします。経路が一方通行ではない無向グラフの場合は，A列のBとCにも1がセットされます。これをC言語でプログラムすると，次のようになります。

```
/* 隣接行列 */
#define N 7
char adjacent[N][N] = {
  0, 1, 1, 0, 0, 0, 0,  /* A */
  1, 0, 1, 1, 0, 0, 0,  /* B */
  1, 1, 0, 0, 1, 0, 0,  /* C */
  0, 1, 0, 0, 1, 1, 0,  /* D */
  0, 0, 1, 1, 0, 0, 1,  /* E */
  0, 0, 0, 1, 0, 0, 0,  /* F */
  0, 0, 0, 0, 1, 0, 0   /* G */
}
```

頂点AからGを数値0から6に対応させるところがポイントです。隣接行列は2次元配列adjacentで表します。内容は図4の隣接行列と同じです。

隣接行列の欠点は，辺の数が少ない場合でもN行N列の行列が必要になることです。つまり，ほとんどの要素が0になってしまい，メモリを浪費してしまうのです。この欠点を補う方法に隣接リストがあります。これは，つながっている頂点を格納する方法です。

A → [B, C]
B → [A, C, D]
C → [A, B, E]
D → [B, E, F]
E → [C, D, G]
F → [D]
G → [E]

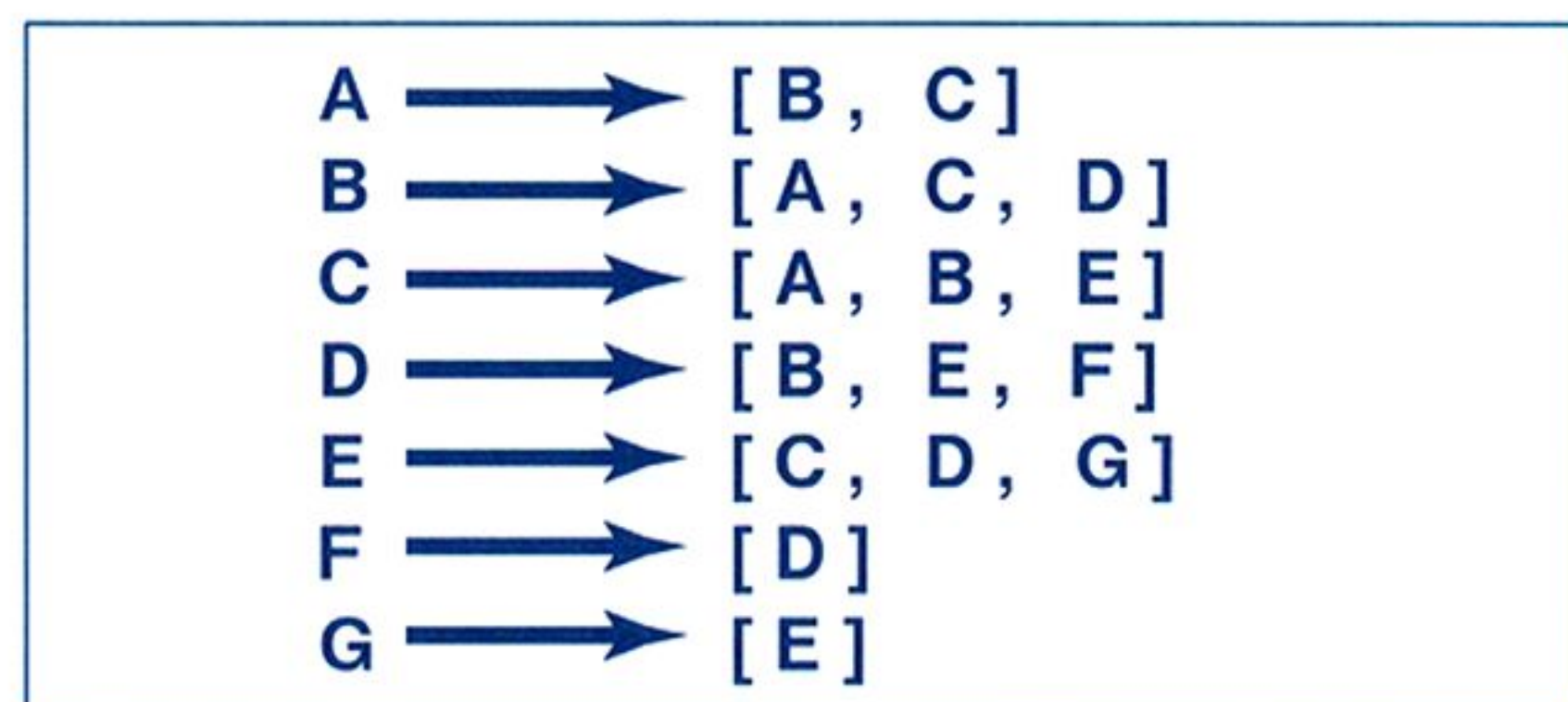



図5　隣接リスト

図5は，頂点とそこに接続されている頂点を→と[ ]で表しています。この表記法は正式なものではなく，Perlのハッシュから拝借したものです。これをC言語で表すと，次のようになります。

```
/* 隣接リスト */
const char adjacent[N][4] = {
  1, 2, -1, -1,  /* A */
  0, 2, 3, -1,   /* B */
  0, 1, 4, -1,   /* C */
  1, 4, 5, -1,   /* D */
  2, 3, 6, -1,   /* E */
  3, -1, -1, -1, /* F */
  4, -1, -1, -1, /* G */
};
```

隣接行列と同様に，頂点AからGを数値0から6に対応させます。－1でデータの終端を表しています。ところで，隣接リストにも欠点があります。たとえば，EとGが接続しているかを調べるには，データを順番に調べていくしか方法がありません。このため，接続の判定に時間がかかることがあるのです。まあ，頂点に接続されている辺の数が少なければ，処理速度が極端に遅くなることはないでしょう。

バックトラックによる探索

今回は隣接リストを使って，AからGまでの経路をバックトラックで求めます。バックトラックを再帰呼び出しで実現する場合，経路を「進む」ことを再帰呼び出しに対応させるのがポイントです。経路を探索する関数をsearchとしましょう。searchは引数として現在地点の頂点を受け取ることにします。最初はsearch（A）と呼び出します。そして，AからBへ進むにはsearch（B）と呼び出します。これでBへ進むことがで

きます。それでは，Aに戻るにはどうしたらいいのでしょう。search (B) はsearch (A) から呼び出されたので，search (B) の実行を終了すれば，呼び出し元であるsearch (A) に戻ることができます。つまり，関数の実行を終了すれば，ひとつ手前の地点にバックトラックできるのです。このように再帰呼び出しを使うと，進むことと戻ることを関数呼び出しで簡単に実現することができます。

それでは具体的に説明します。経路は配列pathに頂点を格納して表すことにします。経路の探索を行う関数searchは，次のように定義します。

```
void search (int len, int node, int goal);
```

引数lenは経路長，nodeは現在地点，goalはゴールを表します。searchはnodeに隣接している頂点をひとつ選び，経路を進めていきます。AからGまでの経路を求めるには，次のように呼び出します。

```
/* A から G までの経路を求める */
search (0, 0, 6);
```

searchはpath[0]にAをセットし，Aに接続されている頂点を選びます。隣接リストから順番に選ぶことにすると，次の頂点はBとなります。Bへ進むためには，次のようにsearchを再帰呼び出しします。

```
/* B へ進むときの再帰呼び出し */
search (1, 1, 6);
```

経路長lenに1を加算することを忘れないでください。この関数の実行を終了すると，呼び出し元の関数である頂点Aの処理に戻ります。プログラムは次のようになります。

```
/* 経路の探索 */
void search (int len, int node, int goal)
{
  path[len] = node;
  if (node == goal) {
    print_path (len);
  } else {
    int n, i;
    for (i = 0; (n = adjacent[node][i]) != -1; i++) {
      if (memchr (path, n, len) == NULL) {
  search (len + 1, n, goal);
      }
    }
  }
}
```

最初に現在地点をpathに格納します。配列pathは大域変数として定義します。次に，ゴールしたかチェックします。これが再帰呼び出しの停止条件になります。ゴールしたらprint_pathで経路を表示します。ここで探索を終了することもできますが，バックトラックすることですべての経路を見つけることができます。パズルを解く場合，解の総数を求めることが多いので，すべての解をもれなく探索する必要があります。バックトラックを使えば，このような要求も満たすことができます。

ゴールしていない場合，隣接リストから次の頂点を選びます。隣接リストから順番に頂点を取り出して，変数nにセットします。このとき，経路に含まれている頂点を選んではいけません。そうしないと，同じ道をぐるぐると回る巡回経路が発生し，ゴールまでたどり着くことができなくなります。標準ライブラリ関数memchrでpath内に頂点nがないことを確認し，searchを再帰呼び出しします。

バックトラックしたときは，同じ経路を進まないために，違う頂点を選ぶことに注意してください。変数iは局所変数なので，関数が実行されている間だけ有効です。たとえば，iが0のときに再帰呼び出しが行われてバックトラックしてきても，iの値は0のままです。それから，for文の更新式i++によってiの値が1となり，隣接リストから次の頂点が取り出されます。つまり，局所変数の働きにより，バックトラックしても同じ頂点が選択されることはないのです。ここでも局所変数が役に立っているわけです。

実行結果は次のようになります。

```
ABCEG
ABDEG
ACBDEG
ACEG
```

4通りの経路を見つけることができました。バックトラックによる探索は，経路を先へ先へ進めるので，「縦形探索」とか「深さ優先探索」と呼ばれています。このため，結果を見てもわかるように，最初に見つかる経路が最短経路とは限りません。最短経路を求めるのに適したアルゴリズムが「幅優先探索」です。これは次回で詳しく説明しましょう。

パズル「マジックスター」

お待たせしました。それでは，実際にパズルを解いてみましょう。コンピュータで解くパズルのなかで，特に有名なのが「8クイーン」です。このパズルはプログラミングの例題に最適なので，教科書や雑誌などで見たことがある人は多いと思います。筆者も月刊・電脳倶楽部に連載されたプログラミング入門講座で取り上げたことがありました。同じパズルを解くのは面白くないので，今回はパズル雑誌でときどき見かける「マジックスター (magic stars)」を解くことにします。

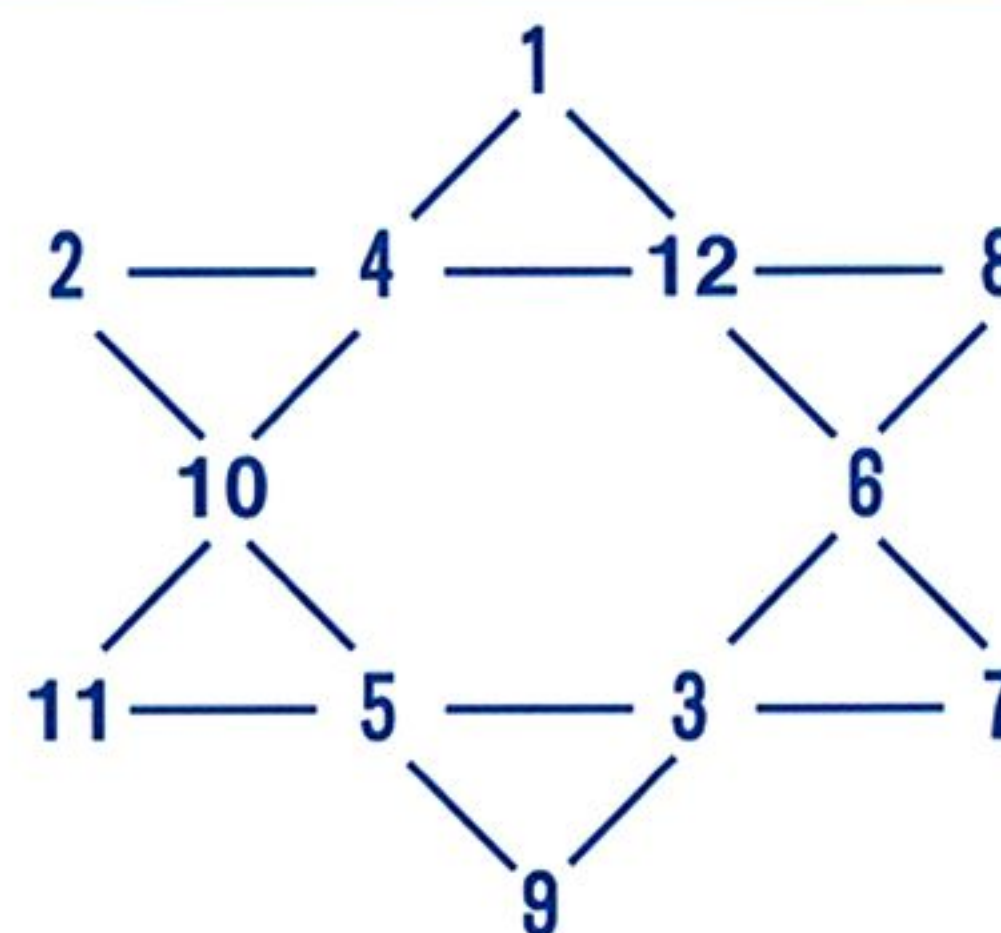
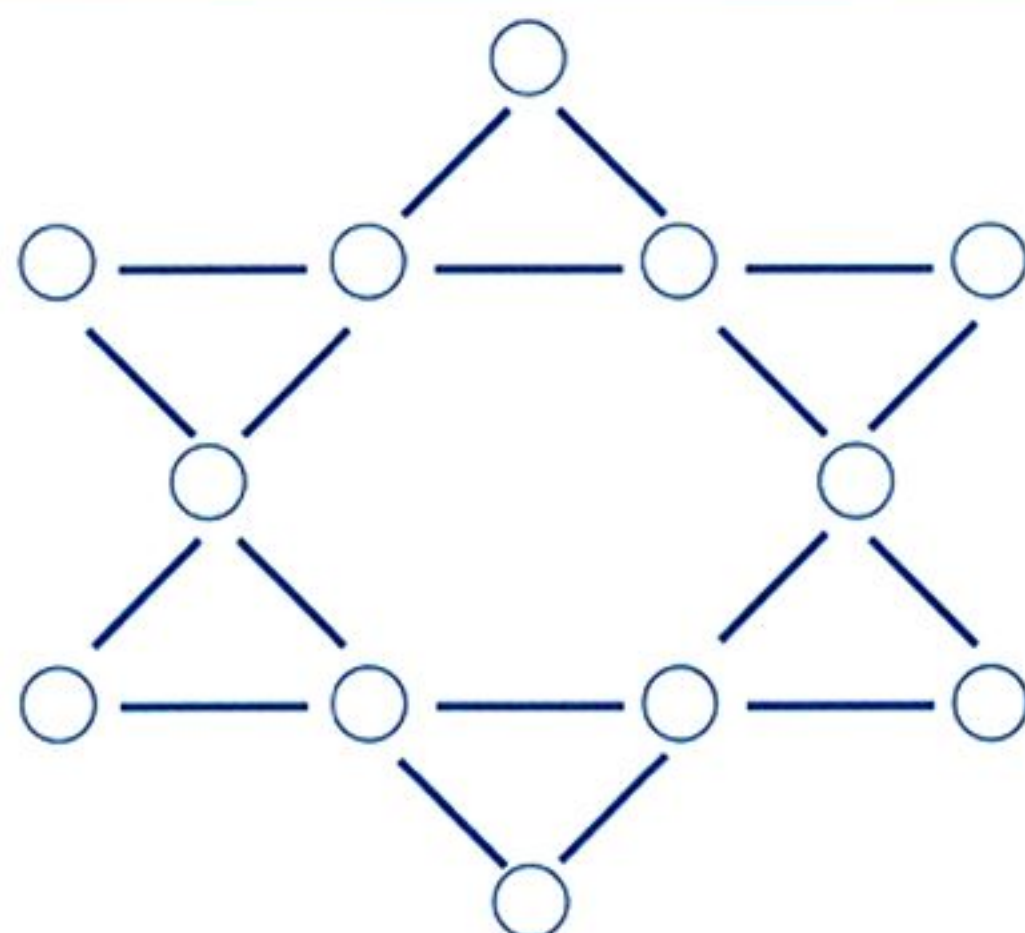
マジックスター (図6) は，12個ある○に1から12までの数字をひとつずつ入れていき，直線上に並んだ4個の数字の合計が，どの直線も26になるような配置を求めるのが目的です。パズル雑誌で出題される場合，ヒントとなる数字がいくつか表示されていて，空いている場所の数字を

正解例

図6　マジックスター

考えるのですが，今回はコンピュータで解くパズルらしく，すべての解を求めることにします。

マジックスターは12個の数字から構成されるので，配列を使って表すことにします。数字は1から12までなので，配列はchar型でいいでしょう。

```
/* マジックスター */
#define N 12
char star[N];
```

配列名はstarとしました。マジックスターと配列starの関係ですが，図7に示すように，○に0から11までの番号をつけ，それを配列の添字に対応させます。

```
              0
            /   \
 1 ──── 2 ──── 3 ──── 4
     \  /         \  /
      5             6
     /  \         /  \
 7 ──── 8 ──── 9 ──── 10
            \   /
              11
```

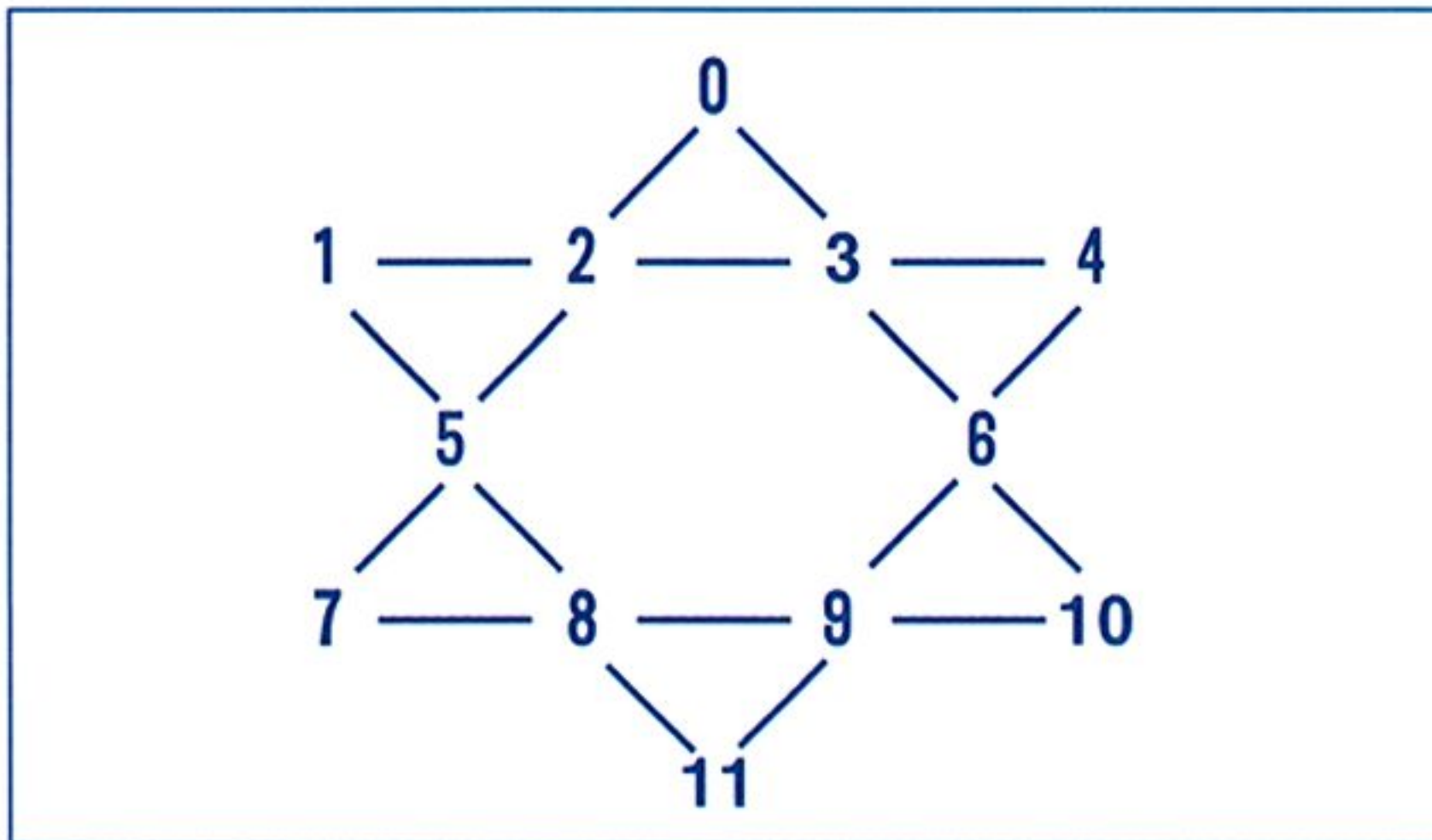



図7　位置を表す番号

つまり，マジックスターの0番の位置に配置された数字は，star[0]の数字と考えるのです。そうすると，6本の直線は配列lineのように表すことができます。

```
/* 直線を表すデータ */
#define LINE  6
const char line[LINE][4] = {
  0, 2, 5, 7,
  0, 3, 6, 10,
  7, 8, 9, 10,
  1, 2, 3, 4,
  1, 5, 8, 11,
  4, 6, 9, 11
};
```

このデータを使って，直線上に並んだ数字の合計を求めることができます。

数字の選択ですが，これは次のように行えばいいでしょう。0番に1を選んだならば，1番にはそれ以外の2から12までの数字から選びます。1番に2を選んだならば，2番には3から12までの数字から選びます。これを11番まで繰り返します。結局，数字の配置は1から12までの順列を求めることと同じになります。要するに，順列を求めて直線上にある4個の数字の合計が26になっているかチェックすればいいわけです。正解の可能性があるデータを生成してチェックする，という方法を「生成検定法（generate and test）」といいます。可能性のあるデータをもれなく作るのにバックトラックは最適です。もちろん，順列を求めるにもバックトラックを使えば簡単です。

順列の生成

それでは，順列を求めるプログラムから作りましょう。まずはウオーミングアップとして，1，2，3の順列を求めてみます。再帰呼び出しを使わないのであれば，次のようなプログラムになるでしょう。

```
#define N    3
#define TRUE  1
#define FALSE 0
char use_number[N + 1];
char perm[N];

void make_perm ( void )
{
  int i, j, k;
  /* 初期化 */
  for ( i = 0; i <= N; i++ ) {
    use_number[i] = FALSE;
  };
  /* 順列の生成 */
  for ( i = 1; i <= N; i++ ) {
  use_number[i] = TRUE;
    perm[0] = i;
      for ( j = 1; j <= N; j++ ) {
      if ( use_number[j] == FALSE ) {
      use_number[j] = TRUE;
      perm[1] = j;
      for ( k = 1; k <= N; k++ ) {
        if ( use_number[k] == FALSE ) {
          perm[2] = k;
          print_perm () ;    /* 順列の完成 */
        }
      }
      use_number[j] = FALSE;
     }
    }
    use_number[i] = FALSE;
  }
}
```

選んだ数字は配列permに格納します。順列は同じ数字を複数回使うことはできません。これをチェックするためにmemchrでpermを検索してもいいのですが，数字の種類が増えると，検索に時間がかかるようになります。そこで配列use_numberを使います。たとえば，1を選んだならばuse_number[1]にTRUEをセットします。あとは，use_numberがFALSEの数字を選んでいくだけです。この方法は経路の探索にも利用できます。

順列が完成したらprint_permで出力します。次の順列を求めるため，使った数字を未使用の状態に戻すことに注意してください。たとえば，順列1，2，3が完成して次の順列を求める場合，2番目のループで数字2を未使用に戻しておかないと，1，3と数字を選んだときに3番目のループで2を選ぶことができず，すべての順列を求めることができなくなります。配列permは上書きされるため，元の状態に戻す必要はありません。

このプログラムは3重のループですが，けっこう大変ですね。1から12までの順列を発生させるとなると，12重のループになってしまいます。ところが，再帰呼び出しを使うと簡単にプログラムできるのです。

```
/* 順列を求める */
void make_perm ( int n )
{
  int i;
  if ( n == N ) {
    print_perm () ;            /* 順列の完成 */
```

```
    } else {
      for (i = 1; i <= N; i++ ) {
        if (use_number[i] == FALSE ) {
          use_number[i] = TRUE;
          perm[n] = i;
          make_perm (n + 1);    /* 再帰呼び出し */
          use_number[i] = FALSE;
        }
      }
    }
  }
```

関数make_permは1からNまでの順列を生成します。Nはマクロで定義します。考え方は経路の探索と同じです。最初の呼び出しでひとつの数字を選び，次の再帰呼び出しで2つ目の数字を選ぶ，というように，N重のループがN回の再帰呼び出しに対応します。再帰呼び出しから戻ってきたら，新しい順列を求めるために，選んだ数字を未使用状態に戻すことを忘れないでください。変数iは局所変数なので，引数nと同様に関数が実行されている間だけ有効です。たとえば，iの値が1で再帰呼び出しが行われたとすると，再帰呼び出しから戻ってきてもiの値は1のままです。このことにより，1からNまでの数字を順番に選ぶことができるのです。

プログラムの骨格は，経路の探索とよく似ていることがわかります。バックトラックによるプログラムは，どのプログラムでもだいたい同じようなかたちになります。基本をしっかりと理解しておけば，バックトラックを自由自在に使いこなすことができるようになります。

マジックスターの解法

あとは，生成した順列がマジックスターの条件を満たしていることを確かめるだけです。これは6本の直線について数値を足し算して，合計が26になるかチェックするだけです。プログラムは次のようになります。

```
/* マジックスターの検査 */
int check_star (void )
{
  int i;
  for (i = 0; i < LINE; i++ ) {
    int j, n;
    for (j = n = 0; j < 4; j++ ) {
      n += star[ line[i][j] ];
    }
    if (n != 26 ) return FALSE;
  }
  return TRUE;
}
```

6本の直線のうち1本でも26でないものが見つかればFALSEを返します。1から12までの順列を生成したら，check_starを呼び出してマジックスターの条件を満たしているかチェックします。プログラムは次のようになります。

```
/* マジックスターを求める */
void search_star (int n )
{
  int i;
  if (n == N && check_stars () ) {
    print_star ();
  } else {
    for (i = 1; i <= N; i++ ) {
      if (use_number[i] == FALSE ) {
        use_number[i] = TRUE;
        star[n] = i;
        search_star (n + 1);  /* 再帰呼び出し */
        use_number[i] = FALSE;
      }
    }
  }
}
```

順列を生成するプログラムmake_permとほとんど同じですが，順列を生成したらcheck_starを呼び出していることに注意してください。条件を満たしていたらprint_star ()でマジックスターを表示します。今回は単純にstarの内容を表示するだけの味気ないものなので，面白くない方はスターの形になるように表示を工夫してください。あとは，main関数で大域変数use_numberの初期化を行い，search_starを呼び出すだけです。

これでプログラム（ソースファイルms1.c）は完成です。ところが，このプログラムをコンパイルして実行してみると，すべての解を出力するのにとても時間がかかるのです。そこで，ms1.cでは1から始まる順列だけに限定したのですが，それでも，実行時間は172秒（Pentium／166MHz）もかかってしまいます。このとき80とおりの解が出力されたので，解の総数は960とおりあることがわかります。すべての解を出力させるとなると，実行時間は単純計算で約35分もかかることになります。生成する順列の総数は12!=479001600とおりもあるのです。これでは時間がかかるのも当然ですね。

パズルを生成検定法で解く場合，チェックするデータをできるだけ絞り込むことが重要です。単純に考えると，膨大なデータをチェックしなければならないようなパズルでも，そのパズル固有の性質をうまく使うことでデータ数を減らすことができます。

マジックスターの場合，1から12までの順列を生成していますが，明らかに無駄なデータを生成しています。たとえば，1，2，3，4，5まで数字を選んだときの配置は図8のようになります。

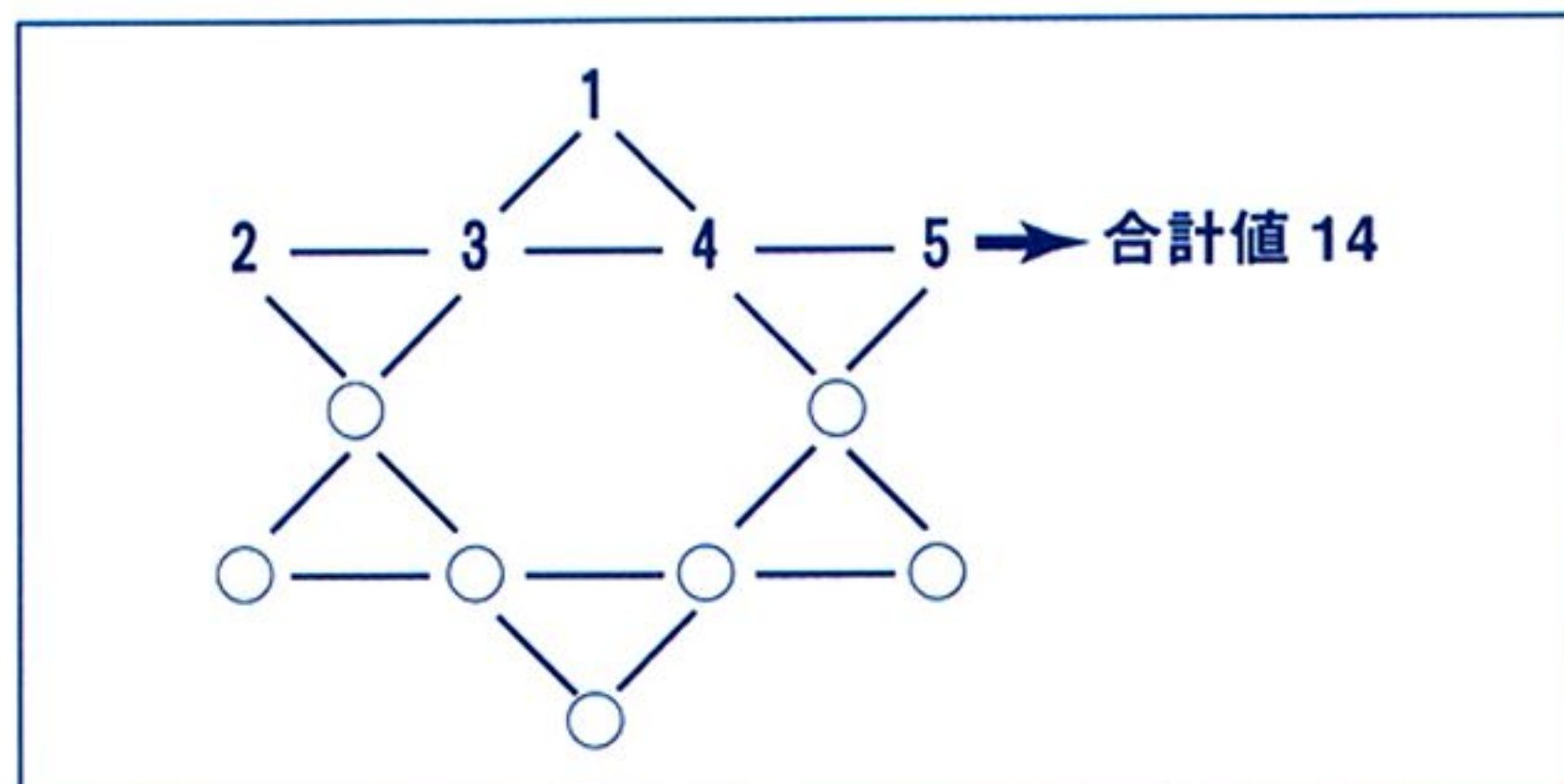



図8　可能性の無いデータ

1本の直線上に4つの数字が並びましたが，その合計値は14にしかなりません。これではマジックスターの条件を満たしませんね。つまり，1，2，3，4，5で始まる順列は，すべて条件を満たさないことがわかるのですが，順列を生成してからチェックする方法では，このような無駄を省くことができません。そこで，数字を配置するときに，直線上に数字が4つ並んだ時点で合計値をチェックすることにします。

このように，できるだけ早い段階でチェックを入れることで，無駄なデータをカットすることを「枝刈り」と呼びます。バックトラックでパズルを解く場合，この枝刈りのよしあしによって実行時間が大きく左右されます。ところが，枝刈りの方法はパズルによって違います。パズル固有の性質をよく調べて，適切な枝刈りを考えることが必要なのです。パズル自体はコンピュータに解かせるのですが，枝刈りの条件は私たちが考えるのです。これも「パズルの解法」の面白いところでしょう。解を求めるだけでなく，いかに効率のよい条件を見つけて実行時間を短縮するか，ということでも楽しむことができるわけです。

枝刈りによる高速化

ひとつの直線に数字が4つ並んだことを確かめる簡単な方法は，配列starの内容を確認することです。数字が置かれていない状態を0と定義すれば，0より大きい値の個数を数えることで実現できます。ですが，数字を選択するたびに配列starを検索するのでは時間がかかりそうです。そこで，直線ごとに置かれた数字を数えるカウンタを用意することにします。

```
/* 数字を置いた個数 */
int number_count[LINE];
/* 数字の合計 */
int total[LINE];
```

直線は配列lineに定義された順番で区別することができます。たとえば，0番に1を選んだとします。ここは直線の0番と1番に属しているので，number_countの0番と1番の要素をインクリメントし，totalの0番と1番の要素に1を加算します。ここで，4個の数字が並んで合計値が26になったか簡単にチェックすることができます。バックトラックするときは，元の値に戻すことを忘れてはいけません。また，位置から直線を求めるのに配列lineを検索すると時間がかかるので，次に示す配列を用意します。

```
/* 位置から直線を求める */
const char position_line[N][2] = {
  0, 1, /* 0 */
  3, 4, /* 1 */
  0, 3, /* 2 */
  1, 3, /* 3 */
  3, 5, /* 4 */
  0, 4, /* 5 */
  1, 5, /* 6 */
  0, 2, /* 7 */
  2, 4, /* 8 */
  2, 5, /* 9 */
  1, 2, /* 10 */
  4, 5, /* 11 */
};
```

配列position_lineを使えば，位置から該当する直線を簡単に求めることができます。

それでは，プログラムを改造しましょう。数字を選択するときに，直線上にある数字の個数と合計値をチェックします。この処理を関数set_numberで行います。プログラムは次のようになります。

```
/* 数値のセット */
int set_number ( int pos, int num )
{
  int i;
  for ( i = 0; i < 2; i++ ) {
    int line = position_line[pos][i];
    if ( number_count[line] == 3 && total[line] + num != 26 ) {
      return FALSE;
    }
  }
  for ( i = 0; i < 2; i++ ) {
    int line = position_line[pos][i];
    total[line] += num;
    number_count[line]++;
  }
  use_number[num] = TRUE;
  star[pos] = num;
  return TRUE;
}
```

位置posに数字numを置いたときに，直線上に数字が4つ並んで，その合計値が26にならなければFALSEを返します。number_countとtotalの値を更新する前に，チェックを行っていることに注意してください。

次は，数字を取り消す関数remove_numberを作ります。プログラムは次のようになります。

```
/* 数字の削除 */
void remove_number ( int pos, int num )
{
  int i;
  for ( i = 0; i < 2; i++ ) {
    int line = position_line[pos][i];
    total[line] -= num;
    number_count[line]--;
  }
  use_number[num] = FALSE;
}
```

数字を取り消すときは，totalとnumber_countの値も元に戻します。これは簡単なので説明は不要でしょう。最後にsearch_star()を改造します。

```
/* 探索 */
void search_star ( int n )
{
  if ( n == N ) {
    print_star () ;
  } else {
    int i;
    for ( i = 1; i <= N; i++ ) {
      if ( !use_number[i] && set_number ( n, i ) ) {
        search_star ( n + 1 ) ;
        remove_number ( n, i ) ;
      }
    }
  }
}
```

数字を選ぶときにset_numberを呼び出し，元に戻すときはremove_numberを呼び出します。マジックスターの条件チェックは，順列を生成している途中のset_numberで行っているため，順列が完成してから行う必要はありません。

これでプログラム（ソースファイルms2.c）は完成です。実行してみると解の総数は960個で，実行時間は結果をファイルへリダイレクトした場合で約5.5秒（Pentium/166MHz）となりました。

対称解のチェック

ところで，パズルの解法では対称解のチェックが必要になる場合があります。対称解とは，盤面を回転させたり裏返しにすると同じになる解のことで，重複解と呼ぶこともあります。盤面に対称性がある場合は必ず発生します。マジックスターの場合，60度ずつ回転させると同じ形になりますね。つまり，回転すると同じになる解（回転解）を6重に数えていることになります。また，マジックスターを裏返しにすると，図9のような配置になります。

このような解を鏡像解といいます。この鏡像解にも回転解が存在するので，全部で12重に数えていることになります。よって，重複しない解は960/12=80通りになるはずです。

それでは対称解をチェックするようにプログラムを改造してみましょう。まず，回転解のチェックですが，0番で選んだ数字に注目してください。選択した数字が1だとすると，マジックスターを60度ずつ回転していくと，1は1番，7番，11番，10番，4番へと移動していきます。これらは同じ解なのですから，0番でほかの数字を選んだ場合でも，これらの位置では数字1を選ぶ必要はありませんね。要するに，0番に配置したことがある数字は，1，4，7，10，11番に配置しないことで回転解を取り除くことができるわけです。

次は鏡像解のチェックです。図9の2番と3番に注目してください。左右の図で2つの位置が入れ替わっていますね。ある解の2番と3番の数字が4，12だったとすると，鏡像解では逆の12,4になるわけです。この数字の大小関係を限定することで，鏡像解をチェックすることができます。つまり，次の条件star[2]<star[3]を満たす解を求めればいいのです。ほかの位置関係でチェックしてもいいのですが，早い段階でチェックしたほうが，枝刈りとしての効果も高くなるので，この位置を選びました。このように，数字を使ったパズルでは，数字の大小関係を限定することで対称解を排除することができます。

対称解をチェックする関数check_symmetryは次のようになります。

```
/* 対称解のチェック */
int check_symmetry (int pos, int num )
{
  static const char flag[SIZE] = {
    0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1,
  };
  if ((flag[pos] && star[0] > num) || (pos == 3 && star[2] > num))
    return TRUE;
  return FALSE;
}
```

引数posは位置で，numはそこに入れる数字です。配列flagは位置1，4，7，10，11を判定するために使います。flag[pos]が1でnumがstar[0]より小さい場合は，すでに0番に配置したことがある数字です。0番は1から順番に数字がセットされるので，数字の大きさを比較するだけでチェックすることができます。これで回転解を判定することができます。posが3のときは，star[2]との大小関係をチェックします。これで鏡像解をチェックできます。回転解か鏡像解であればTRUEを返します。

あとはsearch_starでcheck_summetryを呼び出すだけです。

```
/* 探索 */
void search_star (int n )
{
  if (n == N ) {
    print_star () ;
  } else {
    int i;
    for (i = 1; i <= N; i++ ) {
      if (!use_number[i] &&
          !check_symmetry (n, i) && set_number (n, i) ) {
    search_star (n + 1 ) ;
    remove_number (n, i ) ;
      }
    }
  }
}
```

これでプログラムの修正（ソースファイルms3.c）は終わりです。実際に実行すると，80とおりの解が出力されます。また，対称解のチェックは枝刈りの効果もあるため，実行時間もファイルへリダイレクトした場合で約1.4秒と短縮されます。

このプログラムは単純な枝刈りだけでなので，高速化する余地はまだまだ残っていると思われます。たとえば，数字を選択する順番を工夫すると，もう少し速くなるかもしれません。このプログラムでは，数字を5つ選んだところで1本の直線が完成しますが，数字を4つ選んだ段階で直線が完成するように配置を工夫したほうがいいはずです。また，数字を3つ選んだら残りの数字は自動的に決まります。この数字が12より大きくなったり使用済みの数字であれば，その段階で枝刈りすることがでるはずです。どのくらい速くなるか，興味のある方は改造してみてください。

次回は？

今回説明したバックトラックによる探索は，深さ優先探索とか縦形探索と呼ばれますが，これと対になるのが「幅優先探索」です。この2つがパズルの解法だけではなく，経路の探索のような問題を解くのに使用される基本的なアルゴリズムです。次回は幅優先探索を使って，完成するまでの最短手数を求めるパズルを解いてみましょう。それでは次回をお楽しみに。

参考文献

[1]A.V.Aho，J.E.Hopcroft，J.D.Ullman
「データ構造とアルゴリズム」培風館1987
[2]奥村晴彦
「C言語による最新アルゴリズム事典」技術評論社 1991
[3]近藤嘉雪
「Cプログラマのためのアルゴリズムとデータ構造」 ソフトバンク 1998
[4]広井誠
「続・サルでも書けるCプログラム講座 第5回，第6回，第18回」
月刊・電脳倶楽部 Vol.88，Vol.89，Vol.101 満開製作所

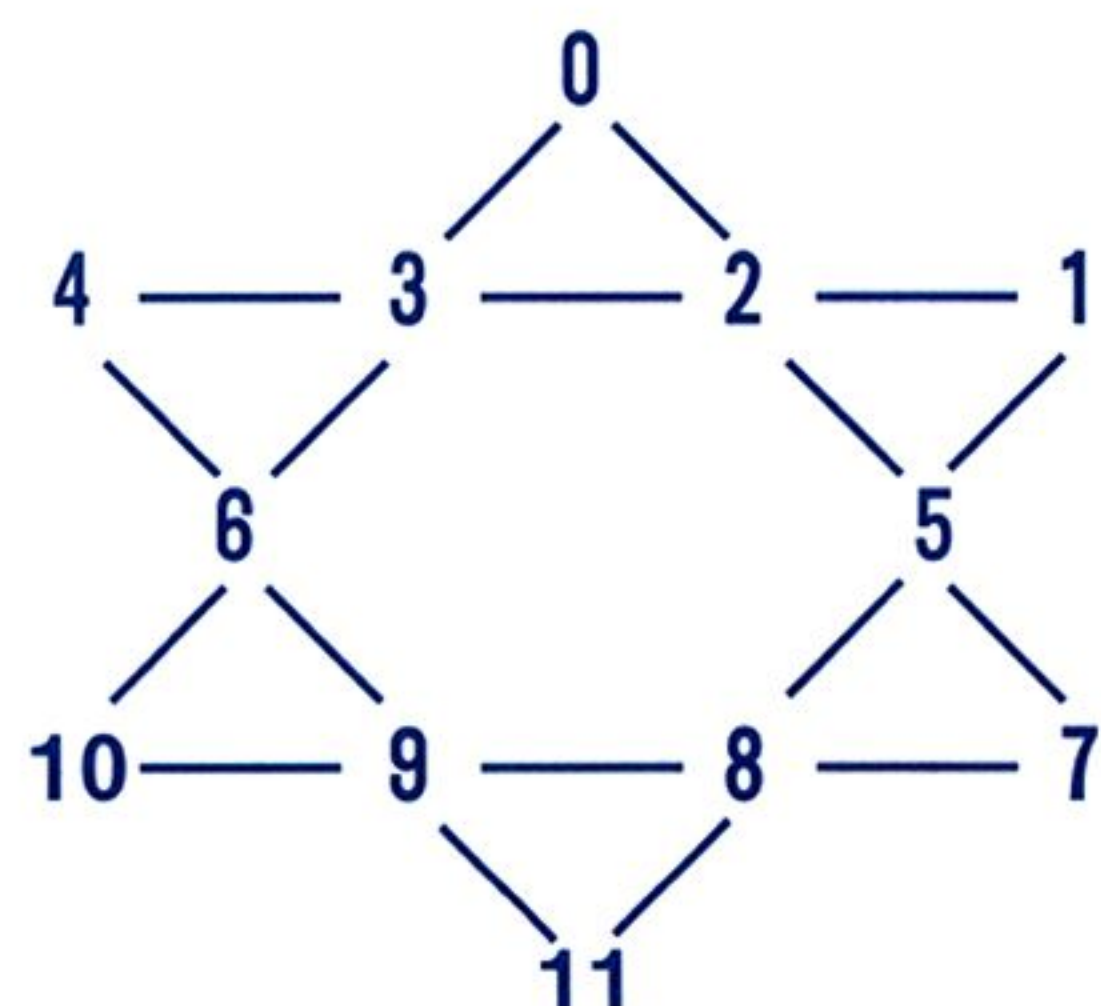
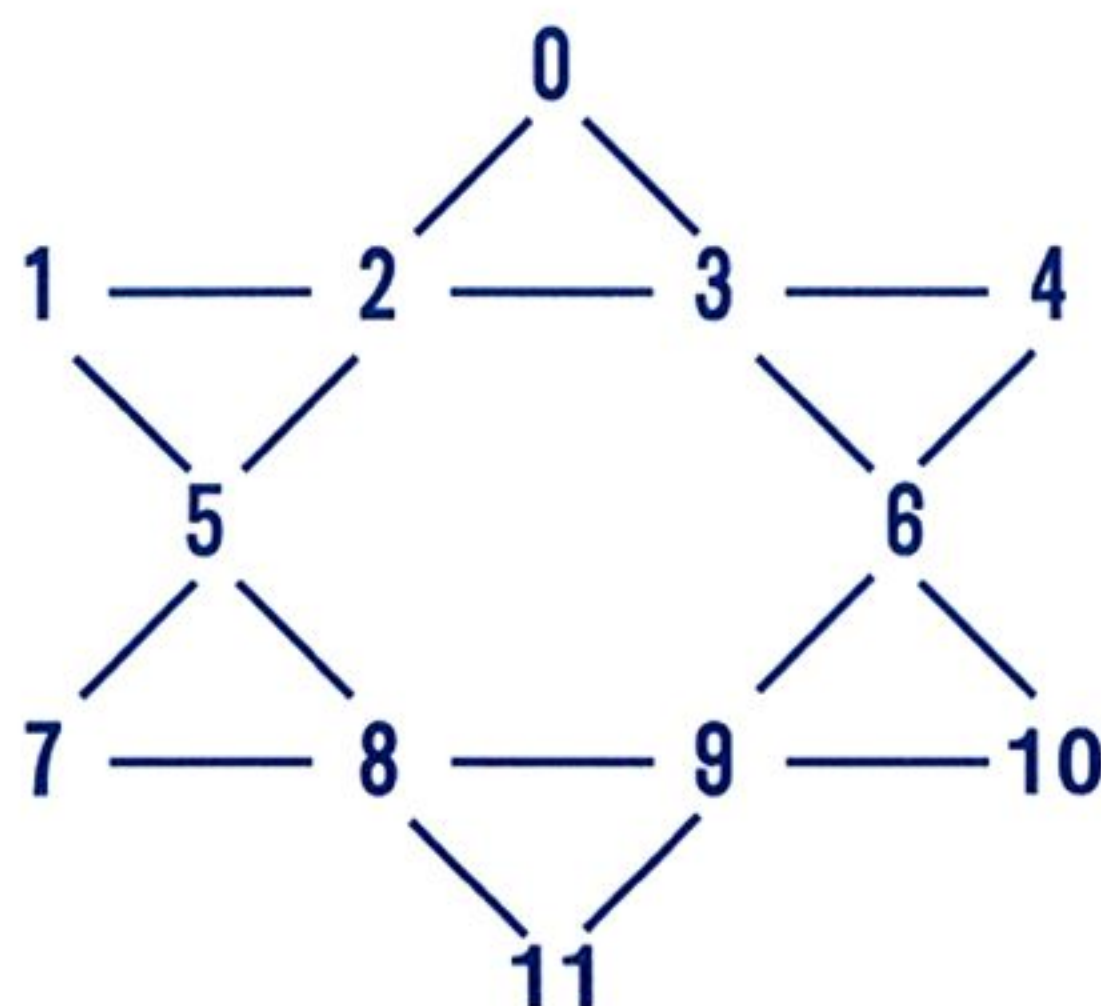



図9 鏡像解

パズルでプログラミング　第2回
幅優先探索と15パズル

広井 誠　Hiroi Makoto

第1回ではパズルの解法に有効なアルゴリズムとして再帰を使ったバックトラックという手法を解説しました。今度は再帰を使わない幅優先探索というアルゴリズムを紹介します。また，15パズル系のパズルの解法プログラムに挑戦してみましょう。

はじめに

前回は再帰呼び出しとバックトラックという，パズルを解くのに必要となる基本的なテクニックを説明しましたが理解できましたでしょうか。再帰呼び出しを前面に押し出したため，再帰嫌いの方にはわかりにくい内容だったかもしれません。ですが，慣れてしまうと再帰呼び出しほど簡単で役に立つテクニックはそうそうありません。これを機会に再帰嫌いの方もそうでない方も，再帰呼び出しをぜひマスターしてください。

今回は再帰嫌いの方でも大丈夫なはずの「幅優先探索」というアルゴリズムを解説します。

幅優先探索

バックトラックによる探索は「深さ優先探索」や「縦形探索」とも呼ばれるように，ひとつの経路を先へ先へと進めていきます。このため最初に見つかる経路が最短経路であるとは限りません。幅優先探索はすべての経路について並行に探索を進めていくため，最初に見つかる経路が最短経路となります。それでは，前回と同じ経路図を使って幅優先探索を具体的に説明しましょう。

B — D — F
A
C — E — G

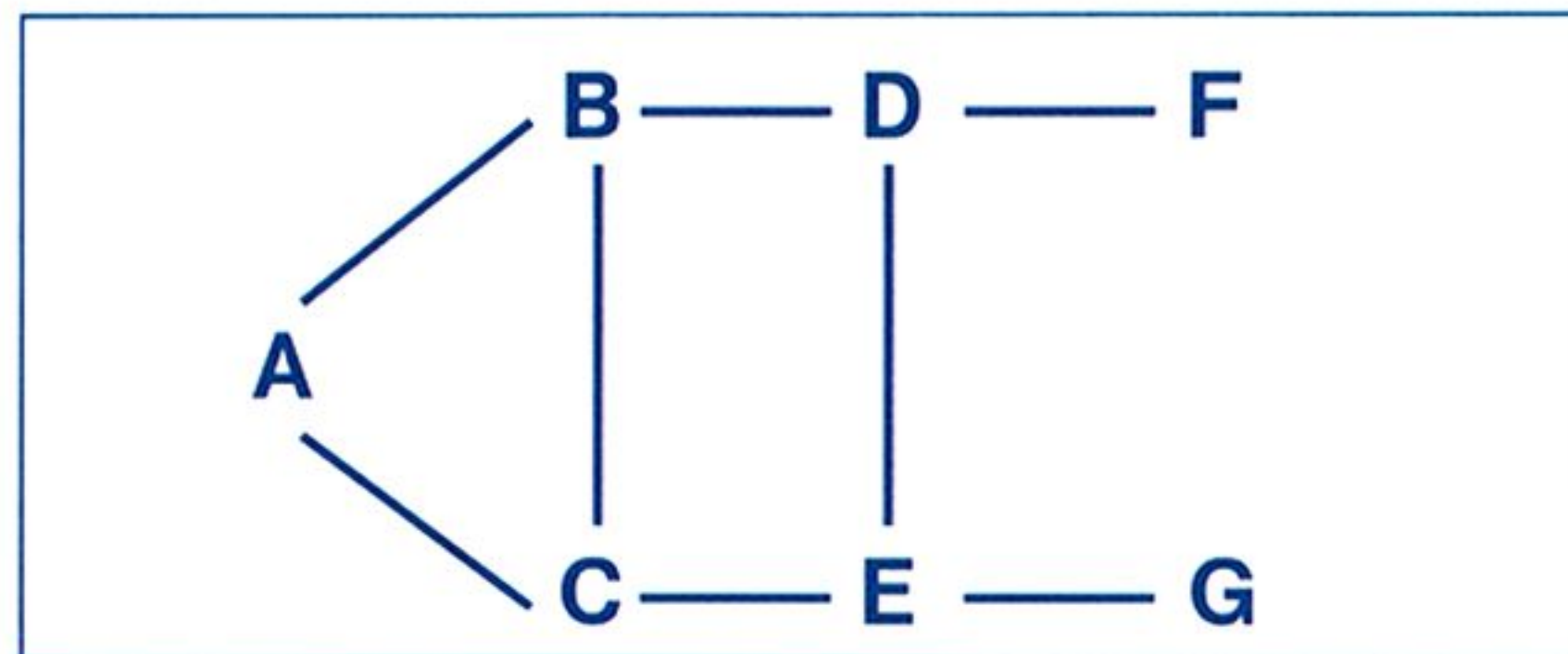



図1　経路図

幅優先探索の様子を図2に示します。

[A] ─ [A,B] ─ [A,B,C] ……
[A,B,D] ─ [A,B,D,F] 行き止まり
[A,B,D,E]
[A,C] ─ [A,C,B] ……
[A,C,E] ─ [A,C,E,G] GOAL
[A,C,E,D]
(出発点)　(2節点)　(3節点)　(4節点)

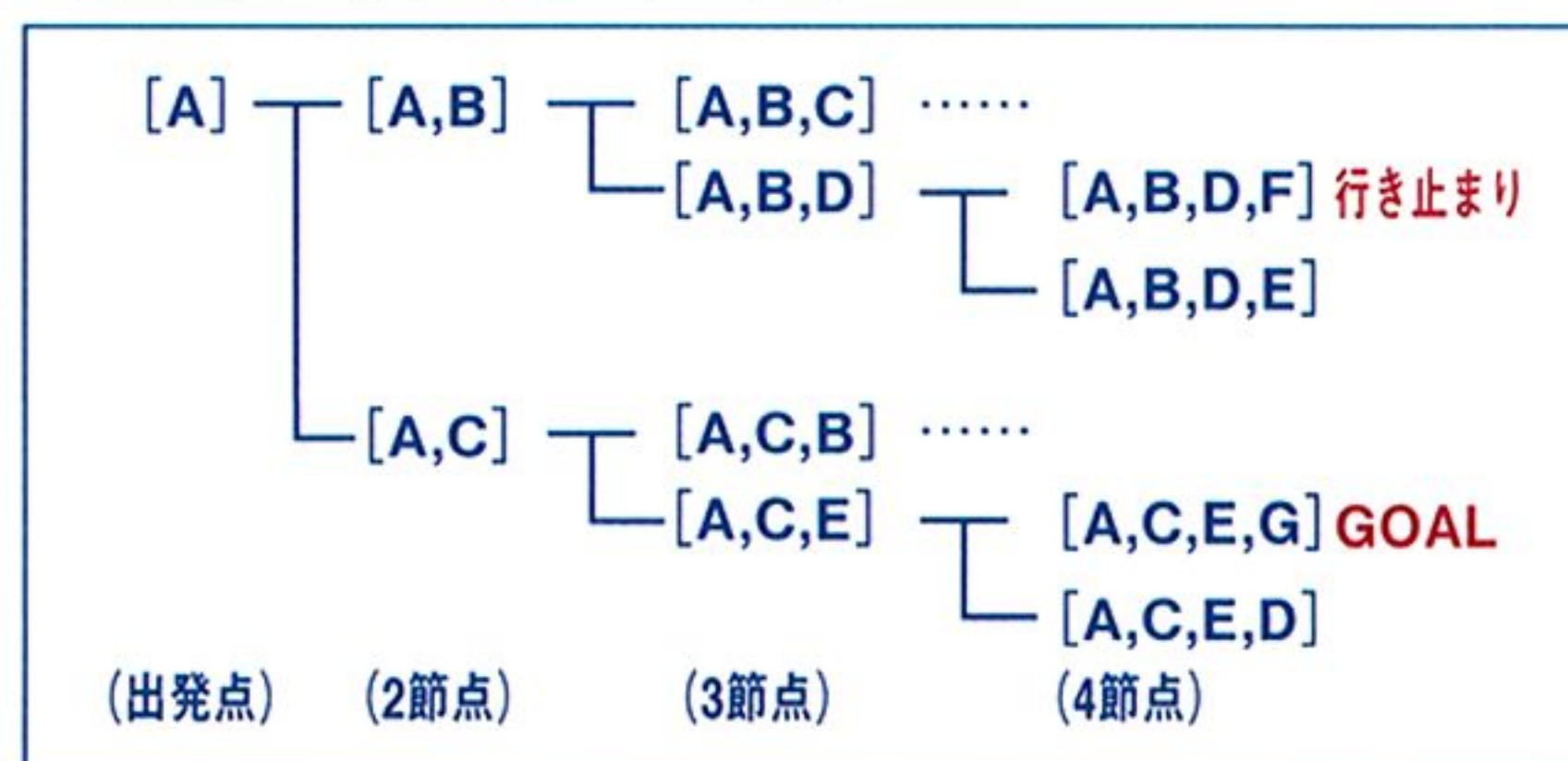



図2　幅優先探索

まず，出発点Aからひとつ進んだ経路（2節点）をすべて求めます。この場合は，[A，B]と[A，C]の2つあり，これをすべて記憶しておきます。次に，これらの経路からひとつ進めた経路（3節点）をすべて求めます。経路[A，B]は[A，B，C]と[A，B，D]へ進めることができますね。ほかの経路[A，C]も同様に進めて，すべての経路を記憶します。あとは，この作業をゴールに達するまで繰り返せばいいのです。

図2では，4節点の経路[A，C，E，G]でゴールに達していることがわかります。このように幅優先探索では，最初に見つかった経路が最短距離（または最少手数）となるのです。この性質は，すべての経路を並行に進めていく探索順序から考えれば当然のことといえるでしょう。このことから，バックトラックの縦形探索に対して，幅優先探索は「横形探索」と呼ばれます。このあとも探索を繰り返せばすべての経路を求めることができます。

完成までの最少手数を求めるパズルを解く場合，幅優先探索を使ってみるといいでしょう。ただし，探索を進めるにしたがって，記憶しておかなければならないデータの総数が爆発的に増加する，つまりメモリを大量消費することに注意してください。

図2の場合では，メモリを大量消費することはありませんが，問題によってはマシンに搭載されているメモリが不足するため，幅優先探索を実行できない場合もあるでしょう。したがって，幅優先探索を使う場合は，メモリの消費量を抑える工夫も必要になります。

スタックとキュー

経路の管理は，「キュー（queue）」というデータ構造を使うと簡単です。たとえば，チケットを買うときには窓口に長い列ができますが，キューはそれと同じだと考えてください。チケットを買うときは，列の途中に割り込むことはできませんね。いちばん後ろに並んで順番を待たなければいけません。列の先頭まで進むと，ようやくチケットを購入することができます。これを表したのが図3です。

このように，キューはデータを取り出すときは列の先頭から行い，データを追加するときは列の後ろへ行います。このため，キューは「待ち行列」とか「先入れ先出し（FIFO：first-in，first-out）」と呼ばれます。

このキューと対になるデータ構造が「スタック（stack）」です。少々脱線しますが，ついでにスタックの動作も説明しましょう。

OUT　　　　IN
← A B C D E F …… Z ←

図3　キューの構造

①空の状態 ②PUSH A ③PUSH B ④POP B ⑤POP A

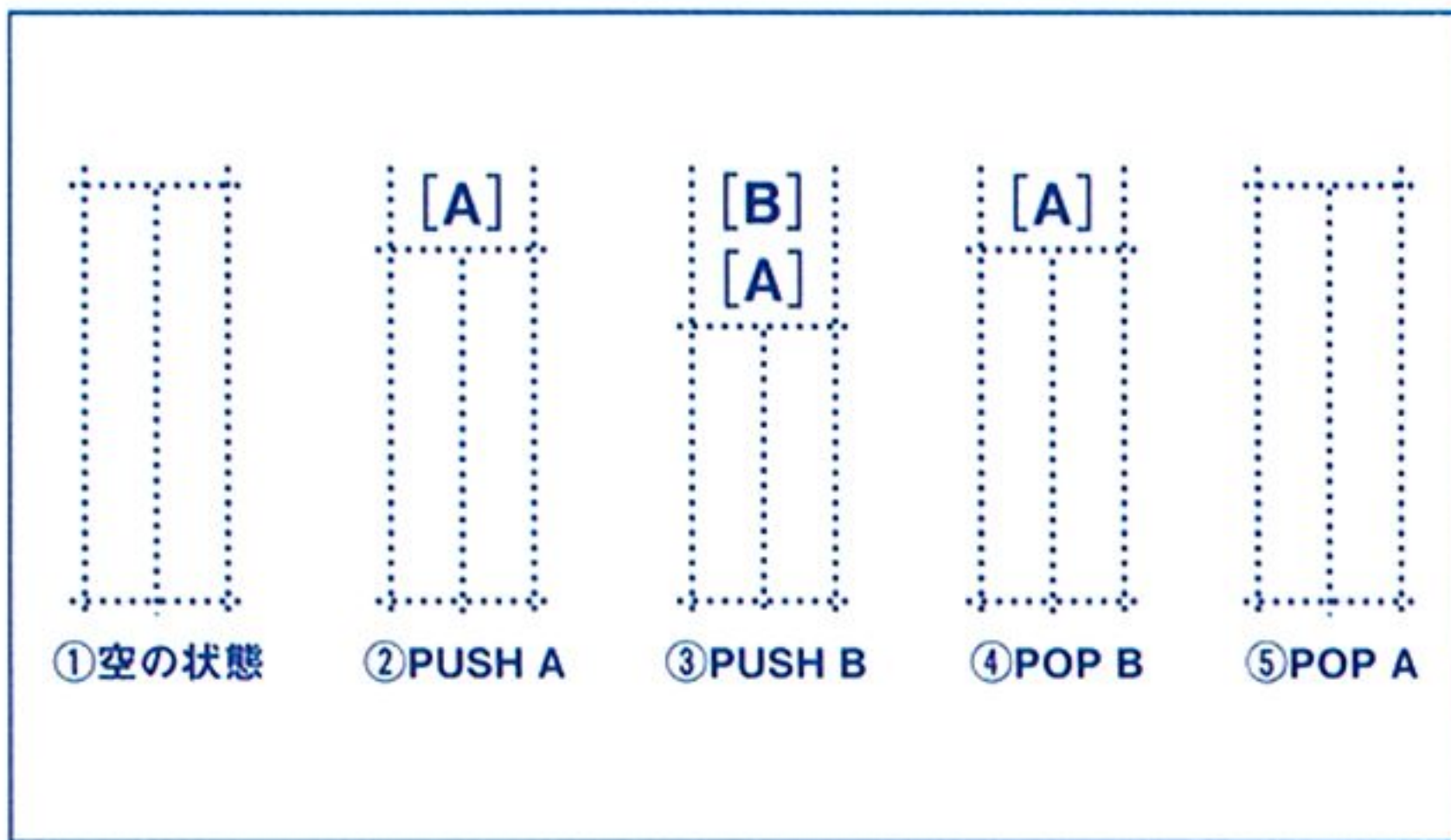



図4 スタックの動作例

図4は，バネがついた容器を表していて，上から品物を出し入れすることができます。初めは空の状態です。ここに品物を載せると，重さによってバネを圧縮し，品物が容器に格納されます。さらにもうひとつ品物を上に載せると，さらにバネを圧縮し，その品物も容器に格納することができます。バネが限界まで圧縮されると，もう品物は追加できなくなります。取り出す場合は，上にある品物から行います。ひとつ取り出すと，その分バネが伸びて下にある品物が上に押し出されます。

この容器の動作が，スタックの動作なのです。スタックにデータを追加する操作をプッシュ（PUSH）といい，スタックからデータを取り出す操作をポップ（POP）といいます。品物をデータに見立てれば，データAをスタックにプッシュし②，次にデータBをプッシュします③。データを取り出す場合，あとから入れたデータBが先にポップされ④，その次にデータAがポップされてスタックが空になります⑤。このように，スタックはあとから入れたデータが先に取り出されるので，後入れ先出し（LIFO：Last-In，First-Out）と呼ばれます。

C言語でスタックとキューを実現する場合には，配列を使う方法がいちばん簡単です。たとえばスタックを実現する場合，データを格納するための配列と，バネの役割を果たすスタックポインタを使います。スタックポインタは，本当にポインタを使ってもいいのですが，添字を表す整数でもかまいません。

まず，配列bufferとスタックポインタspを用意します。spの値は0に初期化しておきます。データをプッシュするときはbuffer[sp]にデータを格納してからspの値をインクリメントします。逆にポップするときは，spの値をデクリメントしてから，buffer[sp]にあるデータを取り出します。スタックを操作するたびに，spの値は図5のように変化します。

データをプッシュしていくと，spの値は配列の大きさと等しくなります。この状態になると，スタックは満杯となります。これ以上データをプッシュすることはできません。また，spが0のときはスタックが空の状態なので，ポップすることはできません。C言語の場合，配列の範囲チェックはプログラマの責任です。実際にプログラムを作るときは，スタックの状態をチェックする処理を忘れないでください。

```
buffer 1  2  3  4  5 sp
(1) [              ] 0 空の状態
(2) [ A            ] 1 PUSH A  buffer[sp++] ← A
(3) [ A  B         ] 2 PUSH B  buffer[sp++] ← B
(4) [ A            ] 1 POPbuffer[--sp] → B
(5) [              ] 0 POPbuffer[--sp] → A
```

図5 配列によるスタックの実現

```
       0 1 2 3 4 5 6 7 8 9
rear = 0 ↓
QUEUE [                    ]：QUEUE は空
front= 0 ↑

rear = 3       ↓
QUEUE [10 20 30            ]：データの追加
front= 0 ↑

rear = 3       ↓
QUEUE [10 20 30            ]：10を取り出す
front= 1   ↑

rear = 3       ↓
QUEUE [10 20 30            ]：20，30を取り出す
front= 3       ↑
```

図6 キューの動作

キューも配列を使って簡単に実現できます。先頭位置を示すfrontと末尾を示すrearを用意し，frontとrearの間にあるデータを，キューに格納されているデータとするのがポイントです。図6を見てください。

まずキューは空の状態で，rear，frontともに0です。データの追加は，rearが示す位置にデータを書き込み，rearの値をインクリメントします。データ10，20，30を追加すると，図6のようにデータが追加されrearは3になります。このときfrontは0のままなので，先頭のデータは10ということになります。

次に，データを取り出す場合，frontの示すデータを取り出しからfrontの値をインクリメントします。この場合，frontが0なので10を取り出してfrontの値は1となり，次のデータ20が先頭になります。データを順番に20，30と取り出していくと，3つしかデータを書き込んでいないので当然キューは空になります。このときfrontは3になりrearと同じ値になります。このように，frontとrearの値が0の場合だけが空の状態ではなく，frontとrearの値が等しくなると，キューは空になることに注意してください。

rear，forntともに値は増加していく方向なので，いつかは配列の範囲をオーバーします。このため，配列を先頭と末尾がつながっているリング状と考え，rear，frontが配列の範囲を超えたら0に戻すことにします。これを「リングバッファ」と呼びます。一般に，配列を使ってキューを実現する場合は，リングバッファとするのが普通です。ですが，今回はリングバッファを使う必要がないので，説明は割愛いたします。興味のある方は参考文献を読んでください。

経路の探索

幅優先探索でのキューの動作を図7に示します。

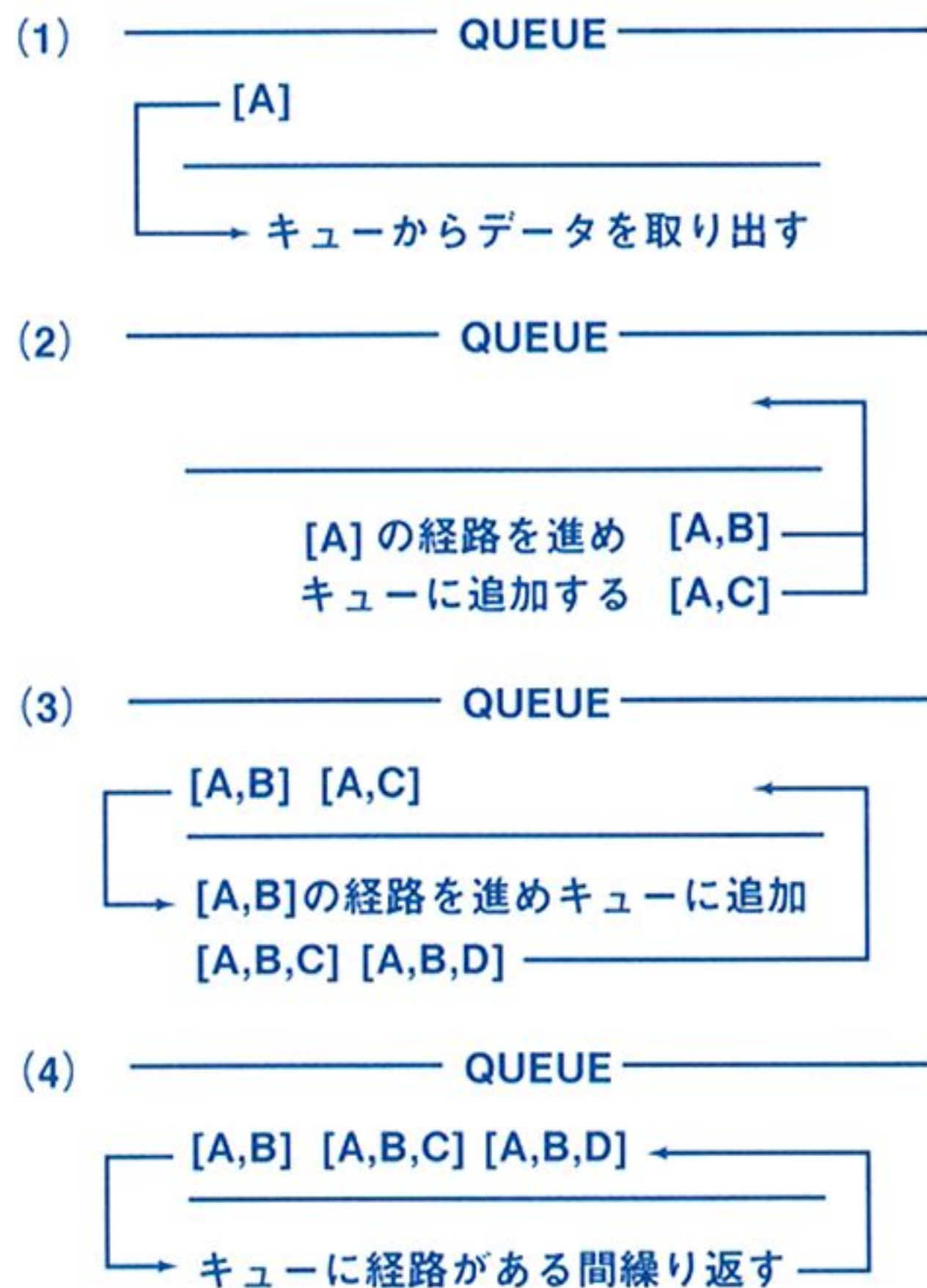



図7　幅優先探索とキューの動作

最初は，(1) のように出発点をキューにセットしておきます。次に，キューから経路を取り出し，(2) のように経路[A]をひとつ進めて，経路[A,B][A,C]を作り，それをキューに追加します。(3) では，経路[A,B]を取り出して，ひとつ進めた経路[A, B, C]と[A, B, D]をキューに追加します。あとはキューに経路があるあいだ処理を繰り返せばいいわけです。

キューは先入れ先出し (FIFO) の性質を持つデータ構造なので，距離の短い経路から順番に処理されるため，幅優先探索として機能するのです。

それではプログラムを作りましょう。経路図は前回と同じく隣接リストで表します。

```
/* 隣接リスト */
#define NODE 7
const char adjacent[NODE][4] = {
  1, 2, -1, -1,   /* A */
  0, 2, 3, -1,    /* B */
  0, 1, 4, -1,    /* C */
  1, 4, 5, -1,    /* D */
  2, 3, 6, -1,    /* E */
  3, -1, -1, -1,  /* F */
  4, -1, -1, -1,  /* G */
};
```

次に経路を格納するキューを定義します。

```
/* Queue */
#define SIZE 64
char path[SIZE][NODE];
char length[SIZE];
```

キューの大きさSIZEは，データ溢れが起こらないように設定します。配列pathに経路を，その経路長を配列lengthにセットします。front，rearは，探索を行う関数searchの局所変数として定義します。プログラムは次のようになります。

```
/* 幅優先探索 */
void search (int start, int goal)
{
  int rear = 1, front = 0;
  path[0][0] = start;
  length[0] = 0;
  while (front < rear) {
    int len = length[front];
    int node = path[front][len];
    int i, n;
    for (i = 0; (n = adjacent[node][i]) != -1; i++) {
      if (memchr (path[front], n, len + 1) == NULL) {
        if (n == goal) {
          print_path (front, goal);
        } else {
          memcpy (path[rear], path[front], len + 1);
          path[rear][len + 1] = n;
          length[rear] = len + 1;
          rear++;
        }
      }
    }
    front++;
  }
}
```

まずスタート地点をキューの先頭にセットします。キューにデータをセットしたのですから，rearの値は1に初期化します。あとは条件front<rearを満たしているあいだは，キューにデータがあるので経路の探索を繰り返します。

次に，キューからデータを取り出し，その経路の先頭の頂点を変数nodeにセットします。これは経路長から簡単に求めることができます。あとはバックトラックと同様に，隣接リストから次の頂点を選び，経路を進めていきます。このとき，経路に含まれる頂点を選ばないことも同じです。

選んだ頂点がgoalであれば，経路をprint_pathで表示します。そうでなければ，経路を進めます。このときはキューから取り出した経路に頂点を追加してはいけません。この経路から複数の経路を作ることになるので，元の経路を書き換えるのではなく，新しく経路を作りそれをキューにセットします。標準ライブラリ関数memcpyで元の経路をrearの位置にコピーし，その経路に新しい頂点を，配列lengthに経路長をセットします。それからrearの値をインクリメントし，新しい経路をキューに追加します。

forループを終了すると，frontをインクリメントし，次の経路を処理します。これで，すべての経路を求めることができます。それでは，実際に実行してみましょう。

実行結果

```
A C E G
A B C E G
A B D E G
A C B D E G
```

結果を見ればおわかりのように，最初に見つかる経路が最短で，最後に見つかる経路が最長となります。当然ですが，経路の総数は4通りとなります。

パズル「おしどりの遊び」

それでは実際にパズルを解いてみましょう。まずはウォーミングアップとして，石を並べ替える「おしどりの遊び」という古典的なパズルを取り上げます。このパズルは囲碁の白石と黒石を交互に並べ，それをペアで動かし

ながら黒石と白石とに分けるというもので，文献[5]によると江戸時代からある遊びだそうです。

黒	白	黒	白	黒	白		

初期状態

黒	黒	黒	白	白	白		

ゴール

図8　おしどりの遊び

石はペアで空いている場所に動かすことができます。このときペアの順番を変えることはできません。たとえば，先頭にある黒白を動かすときに，白黒というように石の順番を逆にすることは許されません。この条件で並べ替えるまでの最短手順を求めます。

盤面は配列で表すことにします。黒石，白石，空き場所をそれぞれマクロでB，W，Sと定義します。

```
/* 種別の設定 */
#define S 0
#define B 1
#define W 2
```

そうすると移動できる石は，連続した2つの場所の値が真であること，で判定することができます。具体的には，次のようなプログラムで移動できるすべての石をチェックすることができます。

```
/* 移動できる駒の判定 */
int i;
for ( i = 0; i < SIZE - 1; i++ ) {
  if ( state[i] && state[i + 1] ) {
    /* 駒を移動できる */
  }
}
```

盤面の状態（局面）を表す配列をstateとし，SIZEは配列の大きさを表すマクロです。石はペアで動かすので，変数iの範囲は0から6までということに注意してください。7までにすると配列の範囲をオーバーしてしまいます。

今度は移動手順の管理を考えましょう。最短手順を求めるだけならば，すべての手順を記憶しておく必要はありません。n手目の移動で作られた局面が，n手目以前の局面で出現しているのであれば，n手より短い手数で到達する移動手順があるはずです。したがって，このn手の手順を記憶しておく必要はないのです。そこで，キューには局面だけを格納し，手順は番号で管理することにします。

	state	prev_state
0	BWBWBWSS	-1
1	SSBWBWBW	0
2	BSSWBWWB	0
3	BWSSBWBW	0
4	BWBSSWWB	0
5	BWBWSSBW	0
6	WBBSSWBW	1
7	WBBWBSSW	1

図9　手順の管理

図9を使って具体的に説明しましょう。局面は配列stateに格納します。このときの添字が，その局面の番号になります。そして，そのひとつ手前の局面の番号を配列prev_stateに格納します。まず最初の配置をstate[0]にセットします。prev_state[0]には終端を表すため－1をセットします。次に，石を移動して1手目の局面を生成します。移動できる石のペアは5種類あるので，新しく生成される局面は5つとなります。それぞれ，state[1]からstate[5]にセットし，prev_stateには元になった局面state[0]の番号0をセットします。

次に，2手目の局面を生成します。state[1]で石を動かして生成される局面は5つありますが，そのうち3つはいままで出現した局面と同じになるので，新しい局面は2つとなります。これをstate[6]とstate[7]にセットします。このときのprev_stateには，元になった局面state[1]の番号1がセットされます。あとは同様に，キューから局面を取り出して石を動かし，新しい局面であればキューに登録することを繰り返します。最終状態と同じ局面になったときは，prev_stateをたどることで手順を再現することができます。

おしどりの遊びを解く

それではプログラムを作ります。最初に，キューの大きさを決めるため石の置き方が何通りあるか数えましょう。これは空き場所の配置から考えたほうが簡単です。2つの空き場所は離ればなれにならないのですから，7通りの配置が考えられます。次に，残り6カ所に3個の黒石を置くことを考えます。これは6個の中から3個を選ぶ組み合わせと考えられるので，組み合わせの公式から(6＊5＊4)/(1＊2＊3)=20通りあります。黒石の置き方が決まれば，白石は残りの3カ所に置くだけです。したがって，全体では20＊7=140通りになるので，キューの大きさは140に設定します。キューの構成は次のようになります。

```
/* キュー */
#define MAX_STATE 140
char  state[MAX_STATE + 1][SIZE];  /* +1 は ワーク領域 */
char  space_postion[MAX_STATE];
short prev_state[MAX_STATE];
```

配列stateは石を動かすときにワーク領域としても使うので，大きさをひとつ余分に設定します。石を動かすには空き場所の位置を求めなければいけません。配列stateを探索してもいいのですが，あらかじめ配列space_postionに空き場所の位置をセットしておけば，簡単に求めることができます。石の移動は次のようになります。

```
/* 石の移動 */
void move_stone ( int front, int rear, int dest )
{
  int j = space_postion[front];
  memcpy ( state[rear], state[front], SIZE ) ;
  state[rear][j] = state[front][dest];
  state[rear][j + 1] = state[front][dest + 1];
  state[rear][dest] = S;
  state[rear][dest + 1] = S;
  space_postion[rear] = dest;
  prev_state[rear] = front;
}
```

引数destは移動する石の位置を表します。space_postionから空き場所の位置を求め変数jにセットします。destとdest+1にある石を，jとj+1へ

移動し，destとdest+1には空き場所を表すSをセットすれば，石を移動することができます。石を動かすときは，state[front]をstate[rear]にコピーしてから行います。これは経路の場合と同じですね。最後に，空き場所の位置をspace_postionに，frontをprev_stateに値を書き込みます。

探索を行う関数searchは次のようになります。

```
/＊ 初期状態 ＊/
const char initial_state[SIZE] = {
  B, W, B, W, B, W, S, S
};
/＊ ゴール ＊/
const char final_state[SIZE] = {
  B, B, B, W, W, W, S, S
};

/＊ 探索関数 ＊/
void search（void）
{
  int front = 0, rear = 1;
  memcpy（state[0], initial_state, SIZE）;
  prev_state[0] = -1;
  space_postion[0] = 6;
  while（front < rear）{
    int i;
    for（i = 0; i < SIZE - 1; i++）{
      if（state[front][i] && state[front][i + 1]）{
        move_stone（rear, front, i）;
        if（!memcmp（state[rear], final_state, SIZE））{
          print_answer（rear）;
          return;
        } else if（!check_same_state（rear））{
          rear++;
        }
      }
    }
    front++;
  }
}
```

プログラムの骨格は経路の探索と同じです。move_stoneで石を動かしたら，ゴールに到達したかチェックします。state[rear]が配列final_stateと同じであれば，解を見つけることができました。print_answerで最短手順を表示します。そうでなければ，同一の局面がないかcheck_same_stateでチェックします。新しい局面であればrearの値をインクリメントして，新しい局面をキューに追加します。move_stoneでstate[rear]にデータをコピーしていますが，rearの値を更新しない限りキューにデータは追加されません。

同一局面のチェックを行う関数check_same_stateは簡単です。

```
/＊ 同じ状態をチェックする ＊/
int check_same_state（int n）
{
  int i;
  for（i = 0; i < n; i++）{
    if（!memcmp（state[i], state[n], SIZE）） return TRUE;
  }
  return FALSE;
}
```

配列stateを先頭から順番にチェックしていきます。これを「線形探索」といいます。このプログラムは次のように書き換えると，ループ中での変数iの範囲チェックを省略することができます。

```
/＊ 番人を使う方法 ＊/
int check_same_state（int n）
{
  int i = 0;
  while（memcmp（state[i], state[n], SIZE）） i++;
  return （i != n） ? TRUE : FALSE;
}
```

検索するデータの最後にはキーデータstate[n]があるので，検索は必ず成功します。発見したデータの位置がnと等しければ，キーデータを見つけたことがわかるので，検索は失敗となります。そうでなければ検索は成功です。キーデータ自身で，検索ポイントがデータの範囲を飛び出さないように監視するわけです。このような方法を「番人」とか「番兵」と呼びます。まあ，最近のように高速CPUを使った環境では，速度の差はほとんどありませんが，番人を使ったほうが簡単にプログラムできるアルゴリズムもあるので，覚えておいて損はありません。

最後に手順を表示するprint_answerを作ります。

```
/＊ 手順の表示 ＊/
void print_answer（int n）
{
  int i;
  if（n != 0） print_answer（prev_state[n]）;
  for（i = 0; i < SIZE; i++）{
    switch（state[n][i]）{
    case S: printf（"空"）; break;
    case B: printf（"黒"）; break;
    case W: printf（"白"）; break;
    }
  }
  printf（"¥n"）;
}
```

prev_stateを順番にたどって出力すると，手順は逆順に表示されてしまいます。そこで，再帰呼び出しを使って最初の状態に戻り，そこから局面を順番に出力させます。

実行結果は次のようになります。

```
黒白黒白黒白黒黒
黒白黒白空白白黒
黒白黒白白空空黒
黒空空白白白黒黒
黒黒黒白白白空空
```

4手で解くことができました。このとき，生成した局面の総数は64個でした。ちなみに，黒と白の分け方を逆にした「白白白黒黒黒空空」も，4手で解くことができます。プログラムは簡単に改造できますが，その前に自分で解いてみるのも面白いでしょう。

6 パズル

次は15パズルでお馴染みの，スライディングブロックと呼ばれるパズルを解いてみましょう。文献[5]によると，15パズルはアメリカのサム・ロイドが1870年代に考案したパズルで，彼はパズルの神様と呼ばれるほど有名なパズル作家だそうです。

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

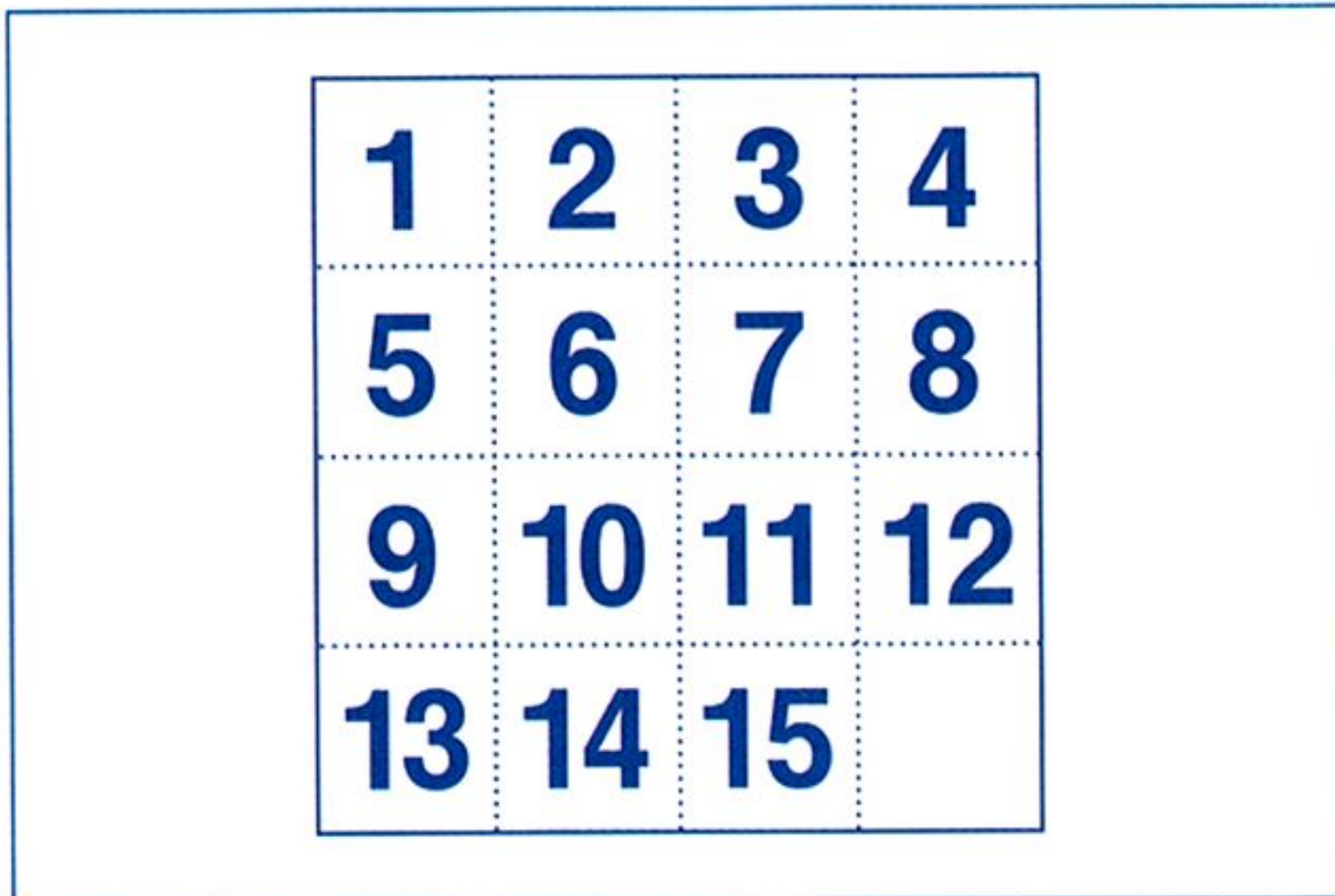



図10　15パズル

15パズルは図10に示すように，1から15までの駒を並べるパズルです。駒の動かし方は，1回に1個の駒を空いている隣の場所に滑らせる，というものです。駒を飛び越したり持ち上げたりすることはできません。

15パズルの場合，駒の配置は空き場所がどこでもいいことにすると，16!(約2e13)通りもあります。実際には，15パズルの性質からその半分になるのですが，それでもパソコンで扱うにはあまりにも大きすぎる数です。そこで，盤面を六角形に変形し，1から6までの数字を並べる「6パズル」を考えることにします。

図11は6パズルをグラフで表したものです。0が空き場所を表します。ここには3，4，6の駒を動かすことができます。6パズルは単純に考えると駒の配置は7!=5040通りとなります。これならば簡単に解くことができそうです。

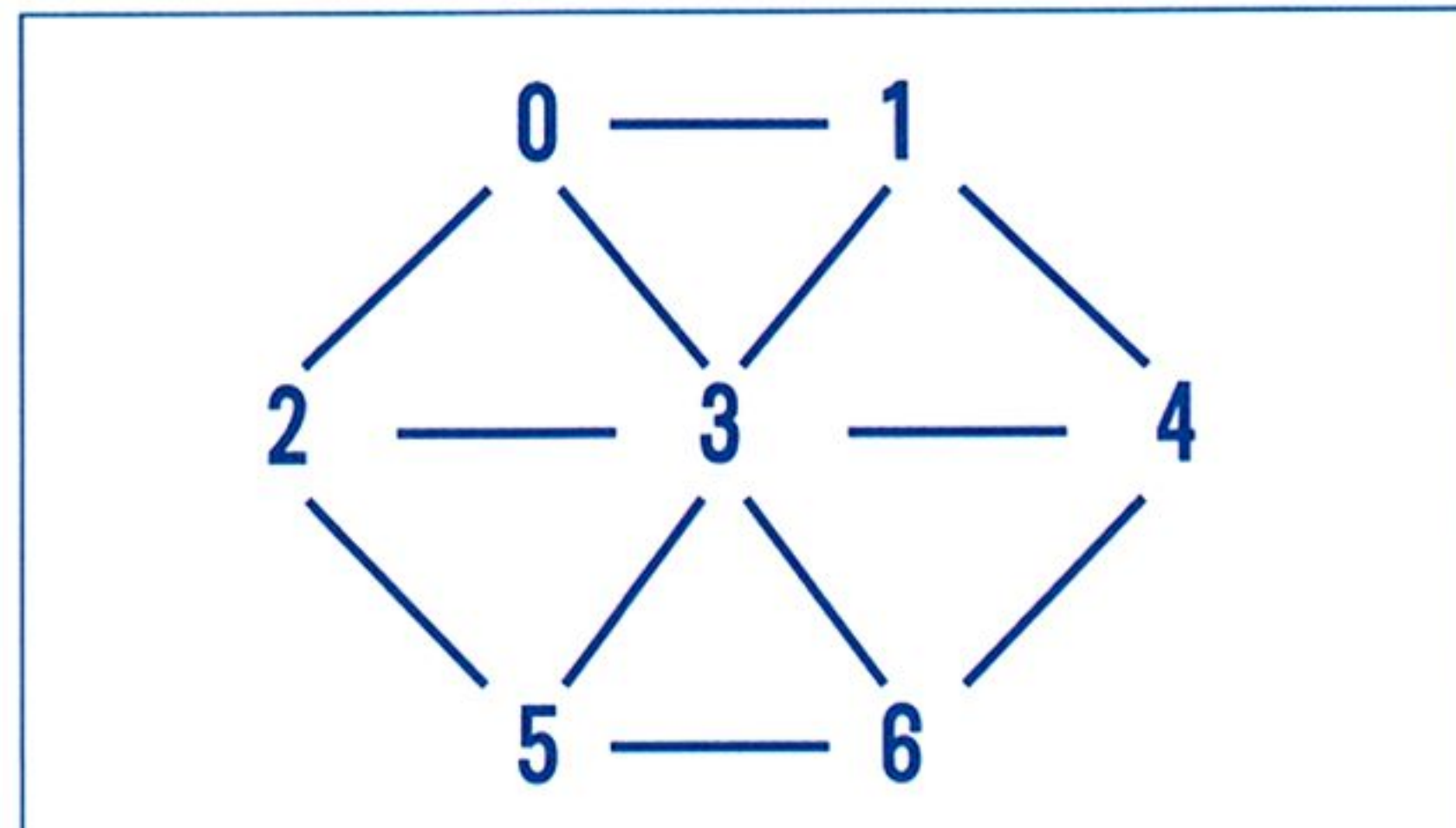
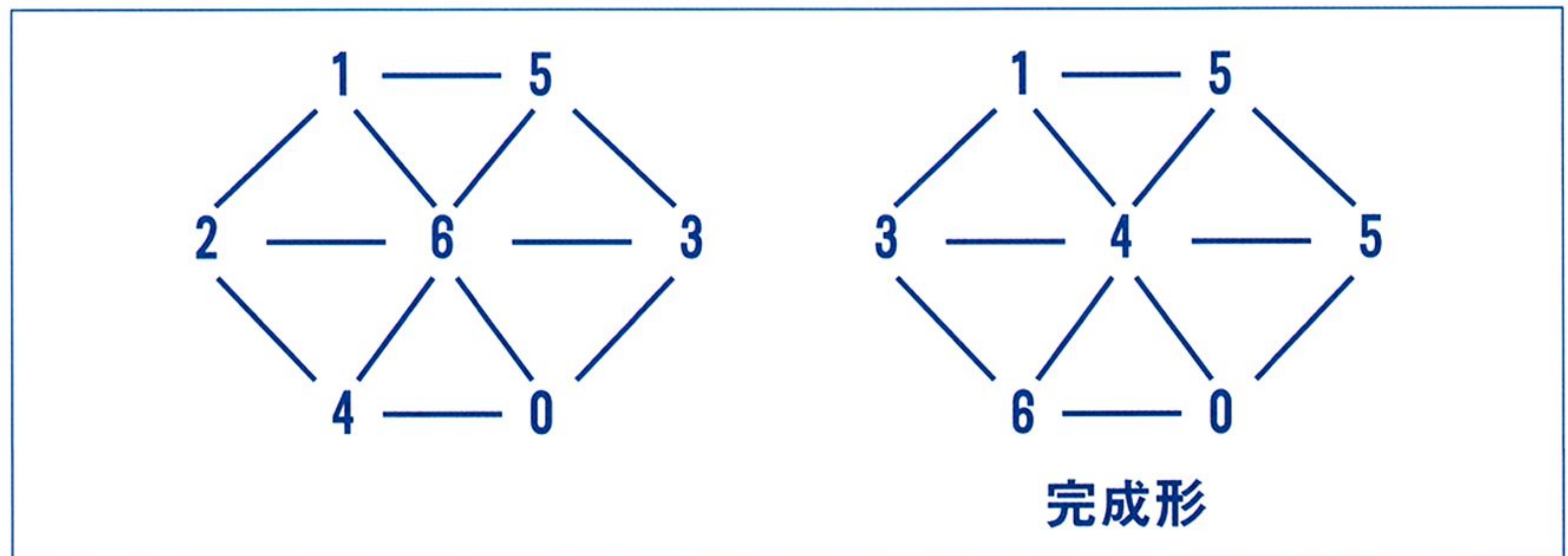



図12　6パズルの位置

6パズルの盤面は配列を使って表します。位置と配列の関係は図12を見てください。

隣接リストとキューの定義は，次のようになります。

```
/* 隣接リスト */
#define SIZE  7
const char adjacent[SIZE][7] = {
  1, 2, 3, -1, -1, -1, -1,  /* 0 */
  0, 3, 4, -1, -1, -1, -1,  /* 1 */
  0, 3, 5, -1, -1, -1, -1,  /* 2 */
  0, 1, 2,  4,  5,  6, -1,  /* 3 */
  1, 3, 6, -1, -1, -1, -1,  /* 4 */
  2, 3, 6, -1, -1, -1, -1,  /* 5 */
  3, 4, 5, -1, -1, -1, -1   /* 6 */
};

/* キュー */
#define MAX_STATE 5040
char  state[MAX_STATE + 1][SIZE];  /* +1 はワーク領域 */
char  space_postion[MAX_STATE];
short prev_state[MAX_STATE];
```

手順の管理は「おしどりの遊び」と同じです。駒の移動は動かすことができる駒を探すよりも，空き場所を基準に考えたほうが簡単です。その局面での空き場所の位置を配列space_postionに記憶しておきます。新しい局面を作るときは，空き場所に隣接している駒を隣接リストから求め，それを空き場所に移動させればいいわけです。

それではプログラムを作ります。

```
/* 初期状態 */
char init_state[SIZE] = {
  1, 5, 2, 6, 3, 4, 0
};
/* 最終状態 */
char final_state[SIZE] = {
  1, 2, 3, 4, 5, 6, 0
};

/* 探索 */
void search (void )
{
  int front = 0, rear = 1;
  memcpy (state[0], init_state, SIZE ) ;
```


図11　6パズル

```
space_postion[0] = 6;
prev_state[0] = -1;
while ( front < rear ) {
int n, i, s = space_postion[front];
  for ( i = 0; (n = adjacent[s][i]) != -1; i++ ) {
    memcpy ( state[rear], state[front], SIZE ) ;
    state[rear][s] = state[rear][n];
    state[rear][n] = 0;
    space_postion[rear] = n;
    prev_state[rear] = front;
    if ( !memcmp ( state[rear], final_state, SIZE ) ) {
        print_answer ( rear ) ;
      return;
    } else if ( !check_same_state ( rear ) ) {
      rear++;
    }
  }
  front++;
}
}
```

プログラムの骨格は前の2つと同じなので，詳しく説明しなくてもいいでしょう。同一局面のチェックを行うcheck_same_stateは単純な線形探索です。手順を出力するprint_answerは，配列の内容をそのまま出力するだけです。六角形に出力するなど，工夫してみてください。

実際に実行すると（ソースファイルhex1.c），11手で解くことができました。このときの実行時間は約3.2秒（Pentium/166MHz）もかかっています。簡単に解けると思っていたのですが，けっこう時間がかかりますね。

時間がかかる理由のひとつは，同一局面のチェックを行うcheck_same_stateにあります。線形探索は配列の先頭から順番にデータを比較していくため，その実行時間はデータ数に比例します。今回生成された局面は2818個だったのですが，ひとつの局面から複数の局面を生成し，それを検索するのですから，データの比較回数は相当の数になるでしょう。実際に数えてみると7598140回にもなります。これでは時間がかかるのも当然ですね。

6 パズルの高速化

このようなときの常套手段が，線形探索に代えて高速な検索アルゴリズムを使うことです。ハッシュ法や二分探索木など，優れたアルゴリズムを使うことで，実行時間を大幅に短縮することができます。ですが，いきなり使ってみましょう，といわれても困ってしまいますね。そこで，ほかの方法を採用することにします。幅優先探索の場合，出発点から探索するだけではなく，ゴール地点からも探索を行うことで，探索する局面数を減らすことができるのです。

その理由を説明するために，簡単なシミュレーションをしてみましょう。たとえば，1手進むたびに3つの局面が生成され，5手で解けると仮定します。すると，n手目で生成される局面は3のn乗個になるので，初期状態から単純に探索すると，生成される局面の総数は，3+9+27+81+243=363個となります。

これに対し，初期状態と終了状態から同時に探索を始めた場合，お互い3手まで探索した時点で同じ局面に到達する，つまり，解を見つけることができます。この場合，生成される局面の総数は3手目までの局面数を2倍した78個となります。

生成される局面数はぐっと減りますね。局面数が少なくなると，同一局面の検索処理に有利なだけではなく，キューからデータを取り出して新しい局面を作るという根本的な処理のループ回数を減らすことになるので，処理速度は大幅に向上するのです。

それではプログラムを改造しましょう。単純に考えると，2つの探索処理を交互に行うことになりますが，そうするとプログラムの大幅な改造が必要になります。ここは，探索方向を示すフラグを用意することで，ひとつのキューだけで処理することにしましょう。メモリを余分に使うことになりますが，プログラムの改造は最小限で済みます。

```
/* 探索方向を格納する */
 #define FORWARD  0
 #define BACKWARD 1
 char  direction[MAX_STATE];
```

探索方向は配列directionに格納します。初期状態からの探索はFORWARDを，終了状態からの探索はBACKWARDをセットします。探索プログラムは次のようになります

```
/* 探索 */
void search ( void )
{
  int front = 0, rear = 2;
  /* キューの初期化 */
  memcpy ( state[0], init_state, SIZE ) ;
  space_postion[0] = 6;
  prev_state[0] = -1;
  direction[0] = FORWARD;
  memcpy ( state[1], final_state, SIZE ) ;
  space_postion[1] = 6;
  prev_state[1] = -1;
  direction[1] = BACKWARD;

  while ( front < rear ) {
    int s = space_postion[front];
    int i, j, n;
    for ( i = 0; (n = adjacent[s][i]) != -1; i++ ) {
      /* ・・・駒の移動・・・（省略） */

      direction[rear] = direction[front];
      if ( (j = check_same_state ( rear )) >= 0 ) {
        if ( direction[j] != direction[rear] ) {
          print_answer ( j, rear ) ;
          return;
        }
      } else {
        rear++;
      }
    }
    front++;
  }
}
```

キューの初期化では，最初に初期状態を，次に終了状態をセットします。2つのデータをセットしたのですから，変数rearの値は2に初期化することに注意してください。最初に，初期状態から1手目の局面が生成され，次に最終状態から1手目の局面が生成されます。あとは，交互に探索が行われます。

駒の移動は同じなので省略しますが，directionの値をコピーする処理を追加していることに注意してください。directionの値を比較するため，check_same_stateは見つけた局面の番号を返すように改造します。見つからない場合は－1を返すことにします。同じ局面を見つけたとき，directionを比較して探索方向が異なっていれば，2方向の探索で同一局面に到達したことがわかります。見つけた最短手順をprint_answerで出力します。同じ探索方向であれば，キューへの追加は行いません。

手順の表示は探索方向によって処理が異なるので，print_answerで振り分けます。

```
/* 結果を出力 */
void print_answer (int i, int j)
{
  if (direction[i] == FORWARD) {
    print_answer_forward (i);
    print_answer_backward (j);
  } else {
    print_answer_forward (j);
    print_answer_backward (i);
  }
}
```

初期状態からの手順を表示する関数がprint_answer_forwardです。この処理は，いままでのprint_answerと同じです。終了状態までの手順を表示するのがprint_answer_backwardです。これはprev_stateを順番にたどって表示するだけなので，繰り返しで簡単にプログラムできます。これでプログラムの改造は終わりです。

プログラム（ソースファイルhex2.c）を実行してみると，生成された局面数は342個で，実行時間は約60msecでした。50倍以上の高速化ですね。予想していた以上の効果に，筆者もたいへん驚きました。

6パズルの最長手順は？

さて，図11の配置は11手で解くことができましたが，5040通りの配置のなかでは，これよりも短い手数で解けるものもあるでしょうし，もっと長い手数がかかるものもあるでしょう。そこで，今度は単純に解くのではなく，パズルが完成するまでにいちばん手数がかかる配置を求めることにします。つまり，最短手順で解いてもいちばん長い手順となる，いちばん難しい配置を求めるわけです。

この場合，5040通りの配置からその最短手順を求めていき，そのなかから最長の手順となる配置を求めることもできますが，それでは時間がとてもかかりそうです。そこで，完成形から始めていちばん長い手数の局面を生成することにします。まず，完成形から駒を動かして1手で到達する局面をすべて作ります。次に，これらの局面から駒を動かして新しい局面を作れば，完成形から2手で到達する局面となります。このように，手数を1手ずつ伸ばしていき，新しい局面が生成できなくなった時点での手数が求める最長手数となります。この処理は幅優先探索を使えばぴったりです。ただし，初期状態からの探索しかできないので，同一局面のチェックが線形探索のままでは時間がかかる，ということは覚悟してください。

このプログラムの目的は，いちばん難しい手順となる配置を求めることなので，手順を表示することは行いません。このため，ひとつ前の局面番号を格納する配列prev_stateは定義しません。その代わり，その局面までの手数を格納する配列moveを用意します。ひとつ前の局面の手数をmoveから求め，それに1を足せば現在の局面の手数となります。

それではプログラムを作ります。

```
/* 探索 */
void search (void)
{
  int front = 0, rear = 1;
  memcpy (state[0], init_state, SIZE);
  space_postion[0] = 6;
  move[0] = 0;
  while (front < rear) {
    int n, i, s = space_postion[front];
    for (i = 0; (n = adjacent[s][i]) != -1; i++) {
      memcpy (state[rear], state[front], SIZE);
      state[rear][s] = state[rear][n];
      state[rear][n] = 0;
      space_postion[rear] = n;
      move[rear] = move[front] + 1;
      if (!check_same_state (rear)) {
        rear++;
      }
    }
    front++;
  }
  print_answer (rear);
}
```

最終状態をチェックする処理がないことに注意してください。生成できる局面がなくなるまで，つまりキューにデータがなくなるまで処理を繰り返します。最後にprint_answerで最長手数とその配置を出力します。この関数は簡単なので説明は省略します。これでプログラム（ソースファイルhex3.c）は完成です。

実際に実行すると，最長手数は15手で，その配置は全部で24通りありました。そのうちのひとつを図13に示します。

ちなみに，生成した全局面は5040個ありました。したがって，6パズルでは数字をランダムに配置しても，必ず完成形に到達できることがわかります。実行時間ですが，予想したように約16秒と時間がかかっています。生成した局面が5040個もあるのですから，データの比較回数は相当の数になります。実際に数えてみると43454059回にもなります。実行時間の短縮には，高速な検索アルゴリズムを使う必要があります。

次回は？

次回はパズルから離れて，高速なデータ検索方法の中から二分探索木を中心に説明します。木構造の基本から多分木まで，ねっとりと説明する予定です。6パズルがどこまで速くなるか，お楽しみに。

参考文献

[1]A.V.Aho，J.E.Hopcroft，J.D.Ullman
「データ構造とアルゴリズム」培風館1987
[2]奥村晴彦
「C言語による最新アルゴリズム事典」技術評論社 1991
[3]近藤嘉雪
「Cプログラマのためのアルゴリズムとデータ構造」 ソフトバンク 1998
[4]広井誠
「続・サルでも書けるCプログラム講座第18回」
月刊・電脳倶楽部 Vol.101 満開製作所
[5]井上うさぎ
「世界のパズル百科イラストパズルワンダーランド」東京堂出版 1997

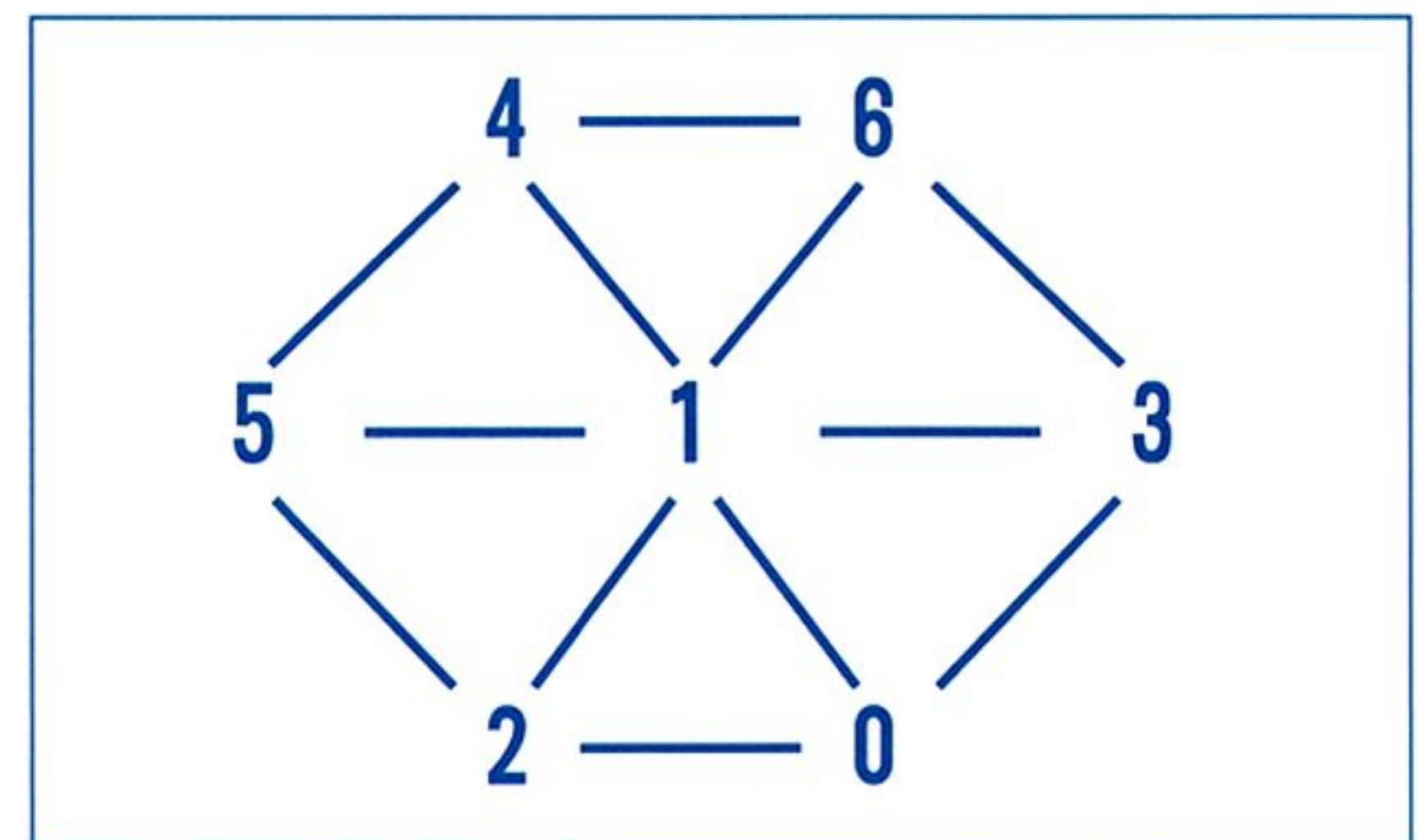



図13　いちばん難しい配置の例

パズルでプログラミング　第3回
二分探索木とハッシュ法

広井 誠　Hiroi Makoto

今回は前回のパズルの高速化を目指してデータチェックの改善を目指します。二分探索，二分木探索などの基本的な概念を押さえていきつつ，ハッシュ関数による実装を行います。さまざまな分野で応用が利く手法ですので，ぜひマスターしてください。

前回説明した「幅優先探索」は，15パズルのように完成するまでの最少手数を求めるパズルを解くのに適しています。幅優先探索を行う場合，生成する局面が多くなるとメモリの消費量のほかに，同一局面のチェック処理が問題になります。単純な線形検索を使うと，ここでの処理に時間がかかるのです。6パズルで最多手数の局面を求めましたが，実行時間がかかったのもこれが原因です。このような場合，データの検索を高速に行うことができれば実行時間を劇的に短縮することができます。

高速なデータ検索といっても難しい話ではありません。私たちの身の回りのことを考えてください。たとえば，筆者の部屋は雑誌や書籍が乱雑に積み上げられていて，とても散らかっています。このような状態では，ある記事を読みたいと思っても，それが掲載されている本を探すのにひと苦労です。整理整頓してあれば簡単に見つけることができるのですが，後悔先に立たずといったところです。まあ，一念発起して整理しても，すぐに散らかってしまうのですが。

これはプログラムの場合にも当てはまる話です。線形探索とは，散らかった部屋の中を力任せに探すことと同じです。データの整理整頓に少々時間がかかっても，探索時間を大幅に短縮できれば，結果として全体の処理時間を短縮できるのです。そこで今回は，高速な検索アルゴリズムの基本である「二分探索木」と「ハッシュ法」を紹介します。ひと言で説明すると，二分探索木は「木」というデータ構造を使ってデータを整理整頓する方法で，ハッシュ法はデータを数値に変換して配列に格納する方法です。

C言語で木構造をプログラムする場合，ポインタを用いる方法が一般的です。また，ポインタを使ってデータ構造を表す場合，もっとも基本的なものに連結リストがあります。連結リストはハッシュ法でも使用するため，最初に連結リストから説明しましょう。そのあとで，木構造，二分探索木，最後にハッシュ法と話を進めていきます。最初は退屈するかもしれませんが，連結リストや木構造は応用範囲が極めて広いデータ構造です。マスターしておけば，今後のプログラミングに役立つことは間違いありません。難しいことはないので，果敢にチャレンジしてくださいね。

連結リスト

連結リスト（linked list）はよく利用されるデータ構造です。配列と同様に複数のデータを格納することができます。C言語の配列は連続したメモリ領域に割り当てられますが，連結リストはデータを一方向につなげることで複数のデータを格納します。C言語では，ポインタを使って連結リストを実現するのが普通です[※1]。まず最初にデータを格納する構造体を定義します。

```
/* セルの定義 */
typedef struct cell {
  int        data;
  struct cell  *next;
} CELL;
```

簡単な例題ということで，連結リストには整数値を格納することにします。連結リストの場合，データを格納する構造体をセル（cell）といいます。セルはデータを格納する変数dataと次のセルを指す変数nextから構成されます。nextが再帰的に定義されているように見えますが，ポインタであることに注意してください。このような構造体を「自己参照構造体」と呼ぶことがあります。nextをポインタではなく，それ自身を含む再帰的な構造にするとコンパイルでエラーとなります。

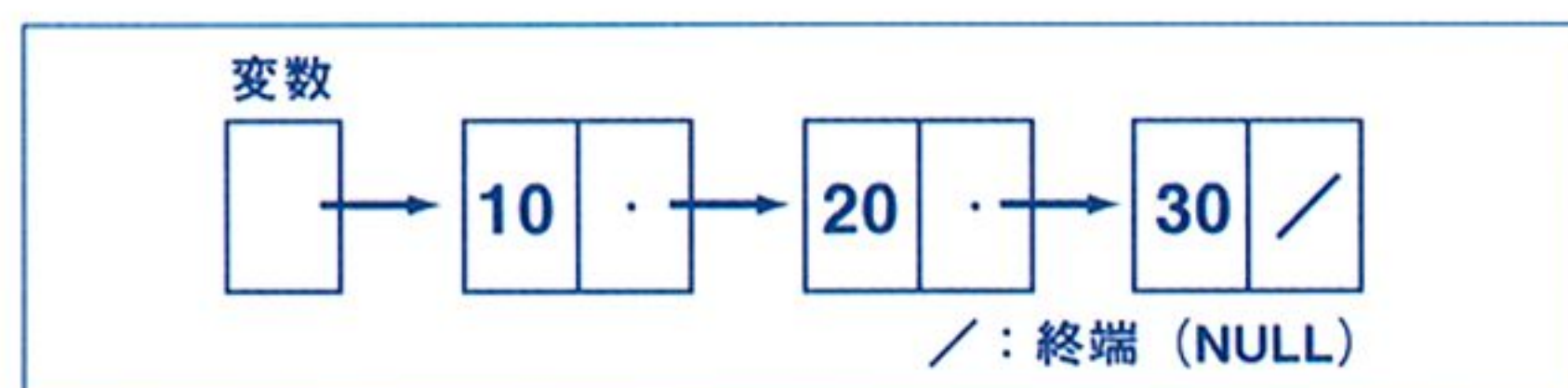



図1　連結リストの構造

図1のように，セルを箱で表すことにします。左側にデータを格納し，右側に次のセルを示すポインタを格納します。このように，C言語ではポインタを使ってデータを連結していく方法がよく使われます。ポインタを使うことで，いろいろなデータ構造を簡単に実現することができるのです。ポインタが苦手な方はこれを機会にマスターすることをおすすめします。

連結リストの終端を表すため，最後尾のセルのnextにはNULLを代入しておきます。データの終端をNULLで表すことも，C言語では一般的な方法です。そして，先頭セルへのポインタを変数に格納しておけば，この変数を使って連結リストにアクセスすることができます。また，データがひとつもない状態を空リストといいますが，変数の値にNULLを代入することで表すことができます。

※1　連結リストは配列を使っても実現できます。この講座でも，最後に配列を使った連結リストが出てきます。

ところで，構造体はデータの構造を定義しただけなので，このままでは使うことができません。プログラムで構造体用のメモリを準備する必要があります。このとき，あらかじめ大域変数として用意しておくこと，取り扱うデータによって用意した変数が足りなかったり，逆に変数が余ってメモリを無駄使いする，といった不都合なことが発生します。このような場合，プログラムの実行中に新しいメモリを取得できると便利です。C言語の標準ライブラリには，自由領域（free store）[※2]からメモリを取得する関数malloc（memory allocate）が用意されています。mallocには取得するメモリのバイト数を与えます。構造体に必要なバイト数はsizeof演算子で求めることができます。したがって，メモリからセルをひとつ取得する関数は次のようになります。

```
/* セルを取得する */
CELL *get_cell ( void )
{
```

```
  CELL *cp;
  cp = malloc (sizeof (CELL) );
  if (cp == NULL) {
    fprintf (stderr, "Out of Memory¥n");
    exit (EXIT_FAILURE);
  }
  return cp;
}
```

mallocが返す値は確保したメモリ領域へのアドレス，つまりセルへのポインタとなります。mallocはメモリを確保できない場合NULLを返します。mallocを使う場合，この返り値を必ずチェックしてください。メモリをたくさん積んでいるから大丈夫だろう，といって省略してはいけません。また，関数fopenでもファイルのオープンに失敗するとNULLを返しますが，このように特別な値を返す関数を使う場合，エラーチェックは欠かせません。プログラムの場合，予期していないことがよく発生するものです。エラーチェックはとても重要なことなのです。

mallocで割り当てたメモリは，不要になった時点でシステムに返すことになっています。これをメモリの解放といい，関数freeを使います。freeで解放されたメモリは再びmallocで割り当てることができます。つまり，メモリを再利用することができるのです。メモリには限りがあるのでmallocで割り当てたまま放置しておくと，いつかは底をついてプログラムは実行不可能となります。mallocを繰り返してメモリを取得する場合，いらなくなったメモリはfreeで解放することを忘れないでください。

freeにはmallocが返したアドレスを与えることに注意してください。それ以外のアドレスを与えた場合の動作は不定です。試したことはありませんが，たぶん暴走することになるでしょう。

ところで，ほとんどの処理系ではプログラムの終了時にmallocで取得したメモリをfreeで解放する必要はありません。プログラムで使用したメモリは実行終了時に解放されますが，このときmallocで割り当てたメモリも解放されるようになっているからです。もちろん，例外的な処理系もあるでしょうが，一般的にはmallocで取得したメモリを最後まで使うのであれば，freeを省略できると考えてもいいでしょう。

※2 昔はヒープ（heap）領域といいました。

連結リストの操作

簡単な例題として，データの探索と挿入を行う関数を作りましょう。最初にデータの探索です。

```
/* データの探索 */
int search (int num, CELL *place)
{
  while (place != NULL) {
    if (num == place->data) return TRUE;
    place = place->next;
  }
  return FALSE;
}
```

連結リストの場合，先頭から順番にデータを比較する線形探索となります。セルをたどる処理ですが，変数placeはセルへのポインタを表していて，次のセルへのポインタはplace->nextに格納されていることに注意してください。したがってplaceを次のセルへ進めるには，place=place→nextとするだけでいいのです。このようにポインタをたどることで，順番に各要素へアクセスすることができます。最後尾のセルの場合，nextにはNULLが格納されているので，placeの値はNULLとなります。これがループの終了条件となります。

次はデータの挿入です。連結リストの場合，先頭にデータを挿入することは簡単です。

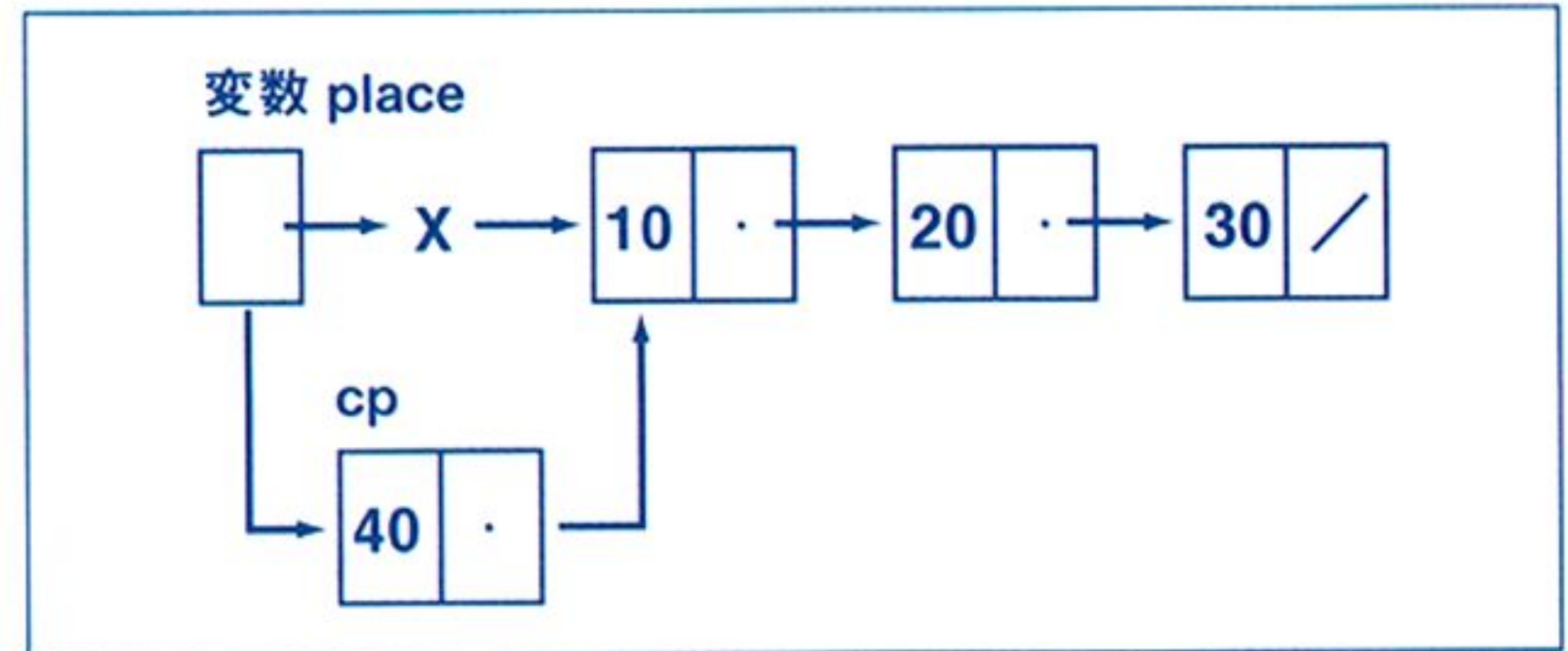



図2 先頭にデータを挿入

連結リストでは，データの挿入はポインタの書き換えで行うことができます。図2を見てください。連結リストの先頭セルは，変数placeにセットされています。変数placeを新しいセルcpへのポインタに書き換え，cpのnextに先頭セルへのポインタをセットすれば，連結リストの先頭にデータを挿入することができます。プログラムは次のようになります。

```
/* 先頭にデータを挿入 */
void insert_top (int num, CELL **place)
{
  CELL *cp = get_cell ();
  cp->data = num;
  cp->next = *place;
  *place = cp;
}
```

変数placeの値を書き換えるため，引数はCELL **placeと宣言して変数のアドレスを受け取ることにします。先頭セルへのポインタはplaceに格納されているので，新しいセルcpのnextにセットしてから，placeの値を書き換えます。順番を間違えるとセルのリンケージが途切れるため，プログラムは動作しなくなります。

空リストにデータを挿入する場合，新しいセルcpが最後尾となります。この場合でも，変数placeに格納されているNULLがcp->nextにセットされるので，プログラムは正常に動作します。逆にいうと，空リストの場合はリストを格納する変数に必ずNULLをセットしてください。

次はリストの途中にデータを挿入する場合です。連結リストの場合，セルの後ろにデータを追加することは簡単に行えます。

図3を見てください。セルc1とc2の間にデータを挿入する場合，c1のnextを新しいセルcpへのポインタに書き換え，cpのnextにc2へのポインタをセットします。c2へのポインタはc1->nextから求めることができます。したがって，c1->nextを書き換える前に，cp->nextへ値をセットします。

それでは具体的にデータを挿入する関数insertを作ります。関数の仕様は，連結リストのn番目にデータを挿入する，ということにします。連結リストの要素は，C言語の配列と同様に0から数えることにします。リストの先頭にデータを挿入する場合は0を指定します。指定した位置が負の値やリストよりも長い場合は，リストの最後にデータを追加することにします。プログラムは次のようになります。

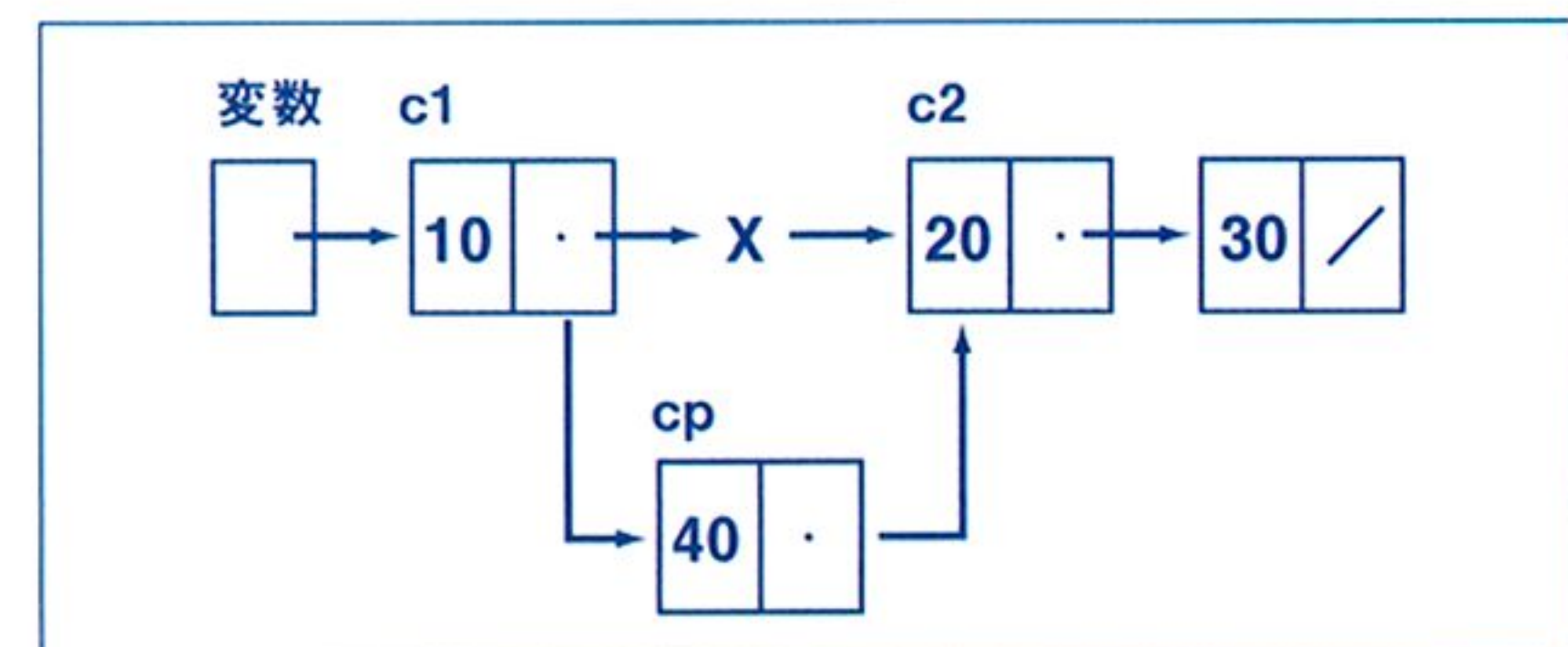



図3 途中にデータを挿入

```
/* データの挿入 */
void insert (int num, int n, CELL **place)
{
 CELL *p, *cp;
 if (!n) {
  insert_top (num, place);
 } else {
  for (p = *place; p->next != NULL; p = p->next) {
   if (! (--n) ) break;
  }
  cp = get_cell ();
  cp->data = num;
  cp->next = p->next;
  p->next = cp;
 }
}
```

引数nが0ならば，insert_topを呼び出してリストの先頭にデータを追加します。そうでなければ，ポインタをたどって目的の位置に移動します。ここで，ループの終了条件に注意してください。nが負の値やリストより長い場合，ループを終了することになりますが，ループの終了条件をp!=NULLで判断すると，ループ終了時のpの値がNULLとなってしまいます。これでは，最終セルへのポインタが失われるため，リストの最後にデータを追加することができません。pが最終セルを指している状態で繰り返しを終了するために，終了条件をpあとnext!=NULLで判断しています。これによりループが終了したあとでも，pの値は最終セルへのポインタとなります。

ループを抜けたあと，変数pが指し示すセルの後ろにデータを挿入にします。まず新しいセルcpを取得し，cp ->dataにはnumをcp ->nextにはp ->nextをセットします。pが最終セルの場合には，p ->nextにNULLがセットされているので，cp ->nextの値は最終セルを表すNULLとなります。最後にp ->nextにcpをセットすれば挿入は終わりです。

連結リストの長所と短所は，配列と比較するとよく理解できます。まず長所から考えてみましょう。配列の場合，宣言した大きさ以上にデータを格納することはできません。連結リストでは，セルを作ることができれば(すなわちメモリがあれば)，データを追加していくことができます。また，途中にデータを挿入する場合，配列では要素を移動する処理が必要ですが，連結リストではポインタの書き換えで済ますことができます。

逆に短所としては，配列のように添字を使って簡単に要素を取り出すことはできません。先頭からn番目の要素を取り出す場合，連結リストでは先頭からポインタをたどっていかなければ，目的地に到達することができません。したがって，ランダムアクセスする場合，配列に比べて時間がとてもかかることになります。もうひとつ，データを格納する場所以外にポインタを格納する場所が必要なので，メモリを余分に使うことも欠点です。

このほかにデータの削除処理がありますが，今回はデータの探索が主題なので説明は割愛いたします。連結リストからのデータの削除は難しい処理ではないので，参考文献を読む前に自分で考えてみるといいでしょう。

木構造

「木(tree)」は，節やノード(node)と呼ばれる要素に対して，階層的な関係(親子関係)を表したものです。身近な例では，ディレクトリ(フォルダ)の階層構造が木にあたります。図4に木構造を示します。

図4ではアルファベットで節を表しています。ディレクトリにルートディレクトリがあるように，木にも「ルート(根)」と呼ばれる節が存在します。図4ではAがルートになります。木を図示する場合，階層関係がはっきりわかるように，ルートを上にして同じ階層にある節を並べて描きます。根からレベル0，レベル1と階層を数えていき，最下層の節までの階層数を「木の高さ」といいます。木は，ある節から下の部分を切り出したものも，木としての性質を持っていて，これを「部分木」といいます。また，節がひとつもない木を「空の木」といいます。

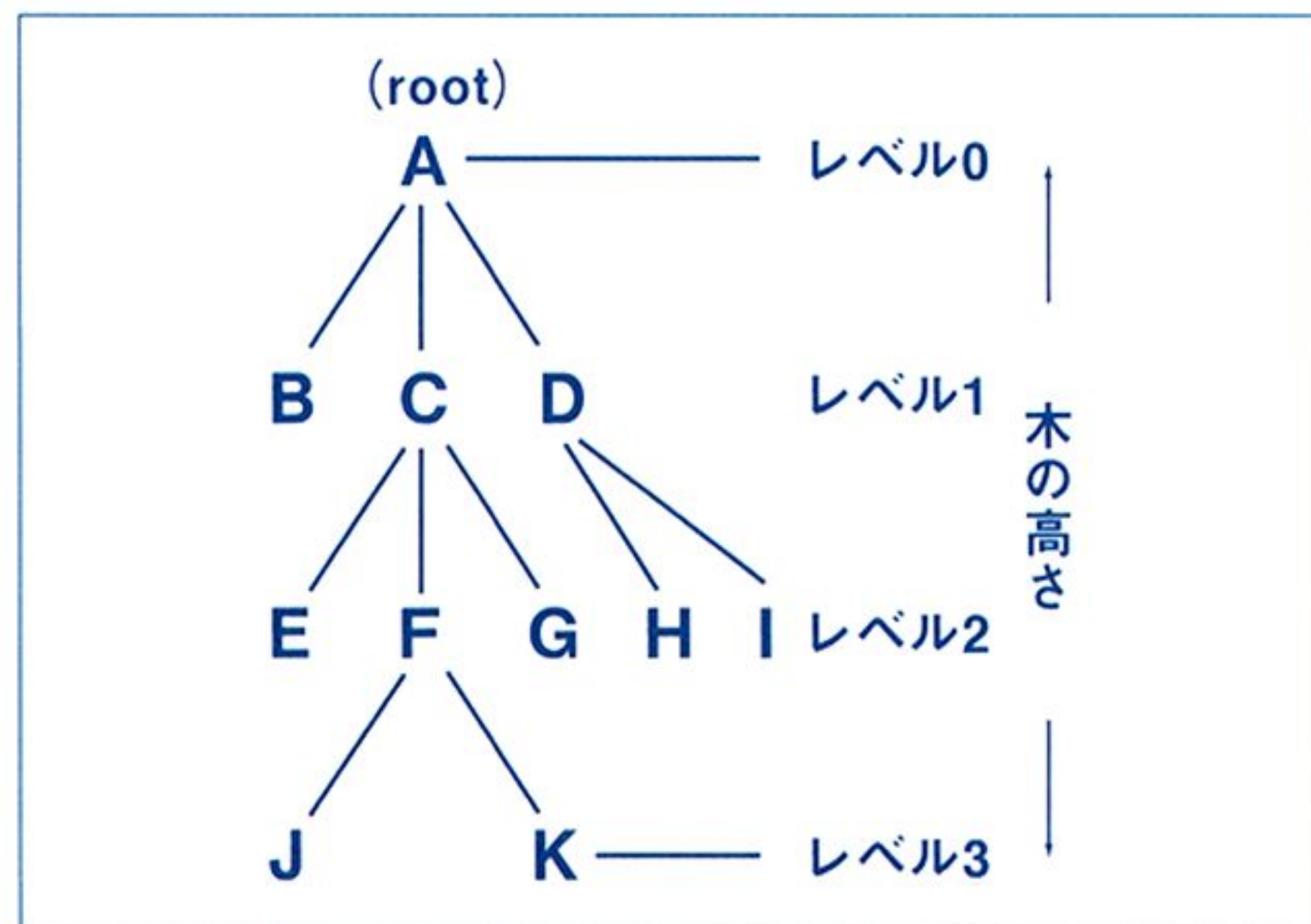



図4 一般的な木構造の一例

木は，ある節からほかの節に至る「経路」を考えることができます。たとえば，AからJには，A－B－G－Jという経路があります。これは，ディレクトリやファイルを指定するときのパスと同じですね。ある節からルートの方向にさかのぼるとき，途中で通っていく節を「先祖」といい，直接つながっている節を「親」といいます。逆から見ると，「子孫」と「子」という関係になります。子を持たない節を特に「葉」ということがあります。図4でいうと，FはJ，Kの親で，JはFの子です。Jは子を持っていないので葉となります。

木構造では，ルート以外の節では必ずひとつの親を持っています。ルートは親を持たない唯一の節です。また，節は複数の親を持つことができません。複数の親を持つ構造は，木ではなくグラフとなります。つまり，木はグラフのなかの特別な構造なのです。

子は「左←右」の順番で節に格納するのが一般的です。これを「順序木」と呼びます。このとき，子の順番を逆にした木は別の木として区別します。また，順番がない木を「無順序木」と呼びます。

節が持っている子の数を「次数」といいます。図4の場合，Aは3つの子B，C，Dを持っているので，Aの次数は3となります。すべての節の次数をnに揃えた順序木を「n分木」と呼びます。特に，次数2の二分木はプログラムでよく使われるデータ構造です。この二分木をデータの探索に用いるのが「二分探索木」です。

二分探索木

二分探索木では，節にひとつのデータを格納します。そして，その節の左側の子には小さいデータを，右側の子には大きなデータを配置するように木を構成します。図5に二分探索木の例を示します。

たとえば，図5から19を探してみましょう。最初にrootの18と探したい数値の19を比較します。19のほうが大きいので右側の子をたどり22と比較します。今度は19のほうが小さいので，左側の子をたどり20と比較します。19のほうが小さいので左側の子をたどり，ここで19を見つけることができました。

二分探索木によるデータの探索は，「二分探索(binary search)」と同じ原理です。二分探索は，あらかじめデータを昇順に並べている配列に対して，特定のデータを高速に探索する方法です。線形探索の実行時間が要素数に比例するのに比べ，二分探索は要素数の対数に比例する時間で探索が完了します。二分探索の過程を図6に示します。

二分探索は探索する区間を半分に分けて調べます。66を探す場合を考えてみましょう。最初に配列の中央値55と66を比較します。データが昇順に整列されていれば，66は中央値55より大きいので前半の区間を調べる必要はなく，後半の区間だけを探索すればよいことがわかります。これと同じことを後半の区間に対して行い，最後には区間に要素がひとつしかなくなり，それとデータが一致すれば探索成功，そうでなければ探索失敗となります。

図5　二分探索木の一例

```
[11 22 33 44 55 66 77 88 99]    66 > 55 後半を探す
             ^
11 22 33 44 55 [66 77 88 99]    88 > 66 前半を探す
                    ^
11 22 33 44 55 [66 77] 88 99    77 > 66 前半を探す
                 ^
11 22 33 44 55 [66] 77 88 99    66 = 66 発見
                ^
```

図6　二分探索

図6の例でも，線形探索を行えば6回繰り返すところが，二分探索であれば4回で済みます。

二分探索木の場合も同じです。左右どちらかの子をたどるたびに，探索するデータ数は半分になります。図5の場合でも，探索するデータ数は15，7，3，1となり最後に見つけることができました。データ数をnとすると，単純な線形探索では平均でn/2回の比較が必要になりますが，二分探索木を使うと，log n程度の回数で収まります。たとえば，100個のデータがある場合，線形探索では50回程度の比較が必要になりますが，二分探索木では7回程度で済むわけです。

どちらも高速に探索できるのであれば，簡単な二分探索を使えばいいじゃないかと思われた方もいるでしょう。ところが，二分探索にはデータの挿入に時間がかかるという欠点があるのです。データが昇順に整列されていないと，二分探索は動作しません。線形探索のように，配列の最後にデータを追加していくことはできないのです。まず，データを挿入する位置を求め，そこから後ろにあるデータを移動する処理が必要になります。挿入位置は，二分探索を使って高速に見つけることができますが，データ数が多くなるほど移動処理に時間がかかるようになります。このため，プログラムの実行中に頻繁にデータの登録を行う処理には，二分探索は向かないのです。

これに対し，二分探索木はデータの挿入も高速に行うことができます。データの探索はルートから始まって下に向かって進んでいきました。データの挿入も同様に，ルートから始まってデータを比較していき，たどるべき部分木がなくなったところにデータを挿入します。

たとえば，図5の二分木に10を挿入してみましょう。根からデータを比較していくと左側の子をたどっていき，11の節にたどり着きます。ここでも左側の子をたどろうとしますが，もう部分木はありません。そこで，11の左側の子として10を挿入します。このとき，配列のようにデータを移動する必要がないため，二分探索木ではデータの挿入も高速に行うことができます。

二分木の実現

連結リストと同じくポインタを使った例題を示します。まず最初に，節を表す構造体を定義します。

```
/＊ 節の定義 ＊/
typedef struct node {
  int         data;
  struct node  ＊left;
  struct node  ＊right;
} NODE;
```

簡単な例題ということで，二分木には整数値を格納することにしました。連結リストと違い，ポインタを格納する変数が2つ必要になります。leftが左側の子，rightが右側の子を表します。子を持たない場合は，連結リストと同様にNULLをセットするのが一般的です。連結リストのように，節を箱で表すと図7のようになります。

連結リストと同様に，ルートへのポインタを変数rootに格納しておけば，この変数を使って二分探索木にアクセスすることができます。また，節がひとつもない空の木は，変数rootにNULLをセットすれば表すことができます。

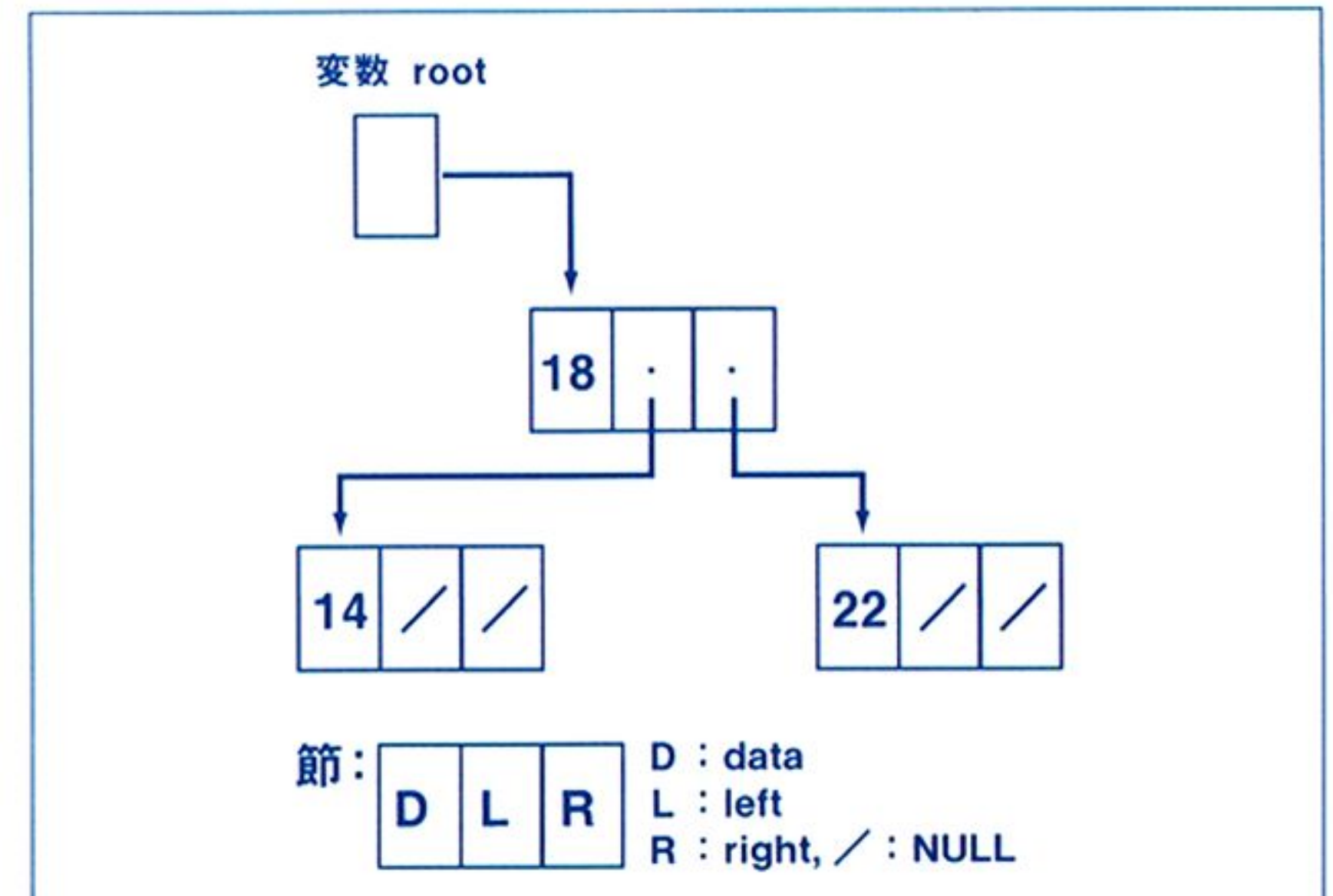



図7　二分探索木の構造

二分探索木の操作

それでは，データを探索する関数から作ってみましょう。この処理は，ルートから始まって下のほうに向かってデータを比較していくだけです。

```
/＊ 探索 ＊/
int search（NODE ＊root, int key）
{
  NODE ＊p = root;
  while（p != NULL）{
    if（key == p->data）{
      return TRUE;
    } else if（key < p->data）{
      p = p->left;
    } else {
      p = p->right;
    }
  }
  return FALSE;
}
```

関数searchには，二分探索木のルートを表す節rootと探索するデータkeyを渡します。ポインタ変数pは，現在探索している節を表します。これはrootで初期化しておきます。あとは，pに格納されているdataを比較し，等しい値であればTRUEを返します。keyが小さいのであれば左側の子をたどり，そうでなければ右側の子をたどります。たどるべき木がなくなればpの値はNULLになるので，whileループを終了しFALSEを返します。二分探索木の動作をそのままプログラムしているだけなので，難しいところはないはずです。

次は，データを挿入する関数を作りましょう。探索と同様に，ルートから始まって下のほうに向かってデータを比較していき，たどるべき木がなくなったところに新しいデータを挿入します。同じデータを見つけた場合はFALSEを返し，データを挿入した場合はTRUEを返すことにします。

```
/＊ 挿入 ＊/
int insert（NODE ＊＊root, int key）
{
  NODE ＊new, ＊＊np = root;
  while （＊np != NULL）{
    if（key == （＊np）->data）{
      return FALSE;
    } else if（key < （＊np）->data）{
      np = &（(＊np）->left）;
    } else {
      np = &（(＊np）->right）;
    }
  }
  new = get_node（）;  /＊ 節をひとつ取得する ＊/
  new->data = key;
  new->left = NULL;
  new->right = NULL;
  ＊np = new;
  return TRUE;
}
```

関数insertは引数としてrootを受け取りますが，最初にデータを挿入する場合，つまり空の木(NULL)にデータを挿入するときは，ルートを格納する変数を書き換える必要があります。このため，引数はNODE ＊＊rootと宣言して，変数のアドレスを受け取ります。変数npも同じく，節を格納する変数のアドレスを表します。最初はルートを格納する変数のアドレスですが，二分木をたどるときは，子を格納しているleftかrightのアドレスを表します。＊npがNULLであれば，たどるべき木がなくなったので，＊npに新しい節を挿入すればいいわけです。もし，空の木であれば＊npはNULLなので，ルートに新しい節が挿入されます。

新しい節は関数get_nodeで取得します。この関数はget_cellと同様にmallocを呼び出してメモリを確保します。メモリ確保に失敗した場合は，エラーメッセージを出力して終了します。あとは構造体のメンバにデータを書き込んで，＊npに新しい節をセットします。

最後に，二分探索木の全データを出力する関数を作ります。二分探索木は，データの大小関係を使って構成されているので，ある順番で節をすべて出力すると，それはソートした結果と同じになります。木のすべての節を規則的な順序で回ることを「巡回(traverse)」といいます。このなかで，次の3つの方法が重要です。

1. 行きがけ順
   まず節のデータを出力，その後左の子，右の子の順番で出力する。
2. 帰りがけ順
   左の子，右の子と出力してから，節のデータを出力する。
3. 通りがけ順
   左の子を出力，次に節のデータを出力，最後に右の子を出力する。

名前の由来は，節のデータを出力するタイミングからきています。節に最初に到達したときに出力する方法が「行きがけ」，子を出力してその節に戻ってきたときに出力する方法が「帰りがけ」，子を出力する途中でその節に戻ってきたときに出力する方法が「通りがけ」です。

二分探索木は「左の子＜節のデータ＜右の子」という関係が成り立つので，通りがけ順に出力すれば，ソートされた出力結果を得ることができます。この処理は再帰定義を使えば簡単に実現できます。

```
/＊ 表示 ＊/
void print（NODE ＊tree）
{
  if（tree != NULL）{
    print（tree->left）;
    printf（"%d¥n", tree->data）;
    print（tree->right）;
  }
}
```

まず，treeがNULLならばなにもしません。これが再帰呼び出しの停止条件となります。あとは通りがけ順の定義そのままにプログラムするだけです。左の子を出力するため，tree ->leftに対してprintを再帰呼び出しします。次に，節のデータtree ->dataを出力します。最後に右側の子を出力するため，tree ->rigthに対してprintを再帰呼び出しします。

このほかに，データの削除処理がありますが，連結リストと同じ理由で説明は割愛いたします。二分探索木の場合，データの探索に比べて削除処理は複雑です。興味のある方は参考文献を読んでください。

6パズル解法の高速化

それでは，二分探索木を使って6パズルの高速化に挑戦しましょう。6パズルの場合，データの総数が決まっているので，二分探索木は配列を使って表すことにします。また，キューとの兼ねあいから配列を使ったほうが簡単にプログラムできます。節とキューの定義は次のようになります。

```
/＊ 節の定義 ＊/
typedef struct {
  char board[SIZE];
  int right;
  int left;
} NODE;

/＊ キュー ＊/
NODE state[MAX_STATE + 1];     /＊ +1 はワーク領域 ＊/
char space_postion[MAX_STATE];
char move[MAX_STATE];
```

配列を使う場合，節の連結はポインタの代わりに配列の添字で表すことができます。したがって，子を格納するleftとrightはintで定義します。子がないことを表す値として，NULLの代わりにNILをマクロで定義します。これは配列の範囲外の値であればなんでもいいのですが，このプログラムでは-1としました。ルートは初期値を格納するstate[0]とします。キューは構造体NODEの配列で表すことができるので，局面の生成はいままでと同じように行うことができます。二分探索木は節の連結により構成されるので，このように配列を使って実現することもできるのです。

次は局面を二分探索木へ登録する関数insert_treeを作ります。二分探索木のなかからstate[i]と同じ局面を検索し，見つからなければ二分探索木へ登録します。プログラムは次のようになります。

```
/＊ 二分探索木への登録 ＊/
int insert_tree（int i）
```

```
{
  int r, n = 0, *np = &n;
  while ( *np != NIL ) {
    r = memcmp ( state[i].board, state[*np].board, SIZE ) ;
    if ( r == 0 ) {
      return FALSE;
    } else if ( r < 0 ) {
      np = & (state[*np].left) ;
    } else {
      np = & (state[*np].right) ;
    }
  }
  state[i].left = NIL;
  state[i].right = NIL;
  *np = i;
  return TRUE;
}
```

節の連結はポインタではなく添字で行っていることに注意してください。変数npは，節を格納する変数rightかleftのアドレスを表します。最初はルートを指し示す変数nのアドレスで初期化します。二分探索木には初期状態の局面が登録されているので，nの値が実際に書き換えられることはありません。このため，nを局所変数として定義しても問題ありません。あとの処理は，例題で示した関数searchと同じです。

最後に幅優先探索を行う関数searchを改造します。

```
/* 探索 */
int search ( void )
{
  int front = 0, rear = 1;
  memcpy ( state[0].board, init_state, SIZE ) ;
  state[0].right = NIL;
  state[0].left = NIL;
  space_postion[0] = 6;
  move[0] = 0;
  while ( front < rear ) {
    int i, n, s = space_postion[front];
    for ( i = 0; (n = adjacent[s][i]) != -1; i++ ) {
      memcpy ( state[rear].board, state[front].board, SIZE ) ;
      state[rear].board[s] = state[rear].board[n];
      state[rear].board[n] = 0;
      if ( insert_tree ( rear ) ) {
space_postion[rear] = n;
move[rear] = move[front] + 1;
rear++;
      }
    }
    front++;
  }
  print_answer ( rear ) ;
}
```

まず初期状態の局面をstate[0]にセットします。leftとrightをNILに初期化することをお忘れなく。あとは同一局面をチェックする関数を新しく作ったinsert_treeに変更します。二分探索木に登録できれば新しい局面なので，それをキューに登録します。変更はこれだけです。

それでは実行結果を表1に示します。

線形探索に比べ実行時間は約50倍の高速化となりました。比較回数も1/100になっています。二分探索木の効果が十分に出ていますね。

表1　6パズルの実行結果 (Pentium/166MHz)

	線形探索	二分木探索
実行時間	約16 s	約330 msec
比較回数	43454059	353230

8パズル

次はもう少し規模の大きい「8パズル」に挑戦してみましょう。

15パズルは4行4列の盤ですが，8パズルは3行3列と盤を小さくしたパズルです。8パズルの場合，駒の配置は空き場所がどこでもいいことにすると，9!= 362880通りあります。ところが15パズルや8パズルの場合，参考文献[4]によると「適当な2つの駒をつまみ上げて交換する動作を偶数回行った局面にしか移行できない」ことが証明されているそうです。図8（2）は7と8を入れ替えただけの配置です。この場合，交換の回数が奇数回のため完成形に到達することができない，つまり解くことができないのです。したがって，完成形に到達する局面の総数は9!/2=181440個となります。

1	2	3
4	5	6
7	8	

（1）完成形

1	2	3
4	5	6
8	7	

（2）完成形に到達できない局面

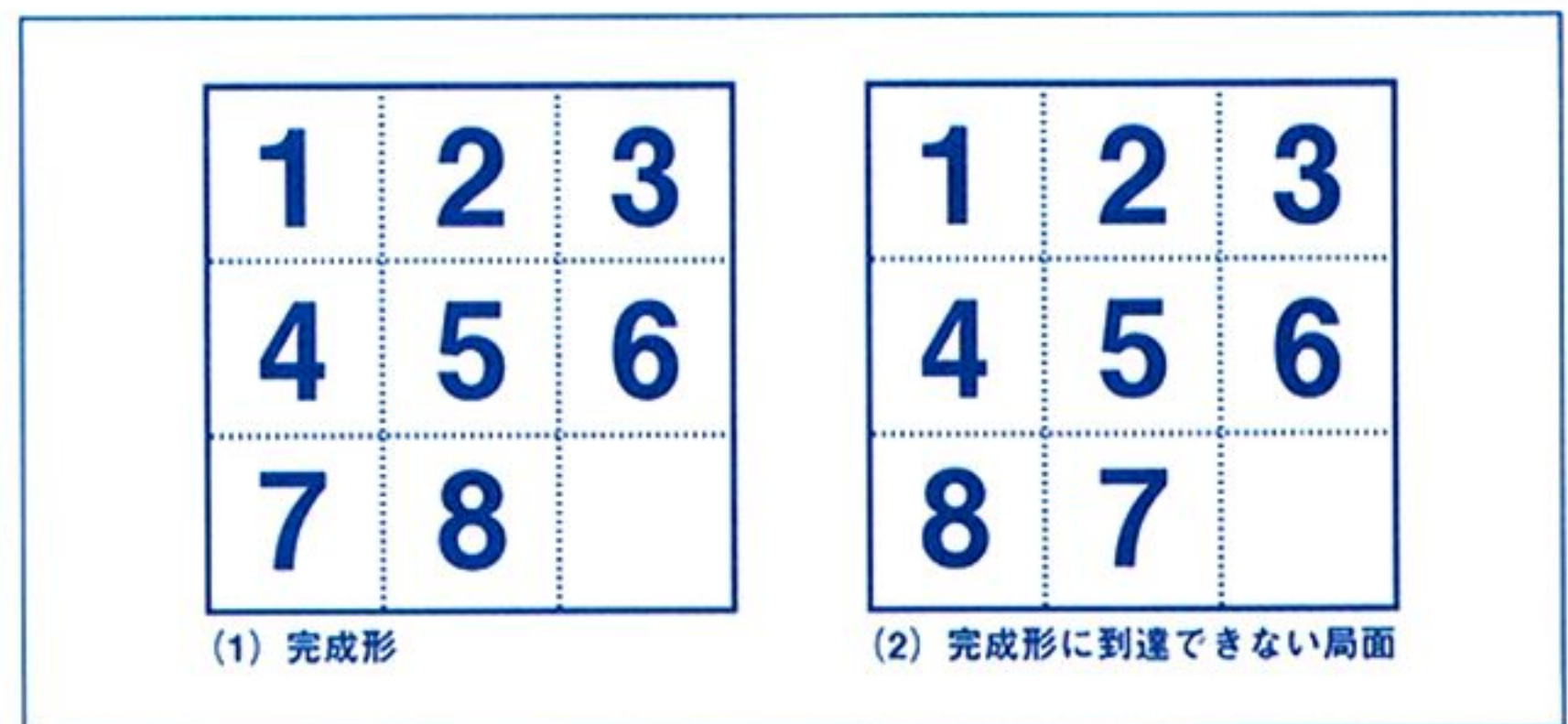



図8　8パズル

これから作るプログラムでは，6パズルと同様に8パズルが完成するまでにいちばん手数がかかる配置を求めることにします。8パズルは6パズルのプログラムを改造することで簡単に作ることができます。まず隣接リストを定義します。座標は図9のように定義しました。

0	1	2
3	4	5
6	7	8

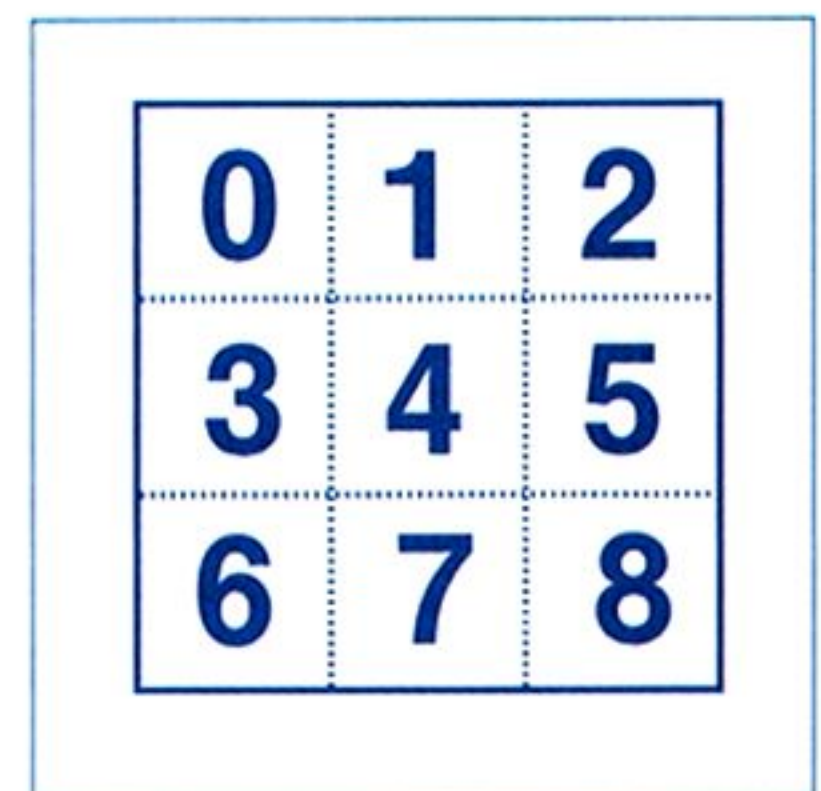



図9　8パズルの座標

```
/* 定数 */
#define SIZE 9
#define MAX_STATE 181440

/* 隣接リスト */
const char adjacent[SIZE][5] = {
  1, 3,-1,-1,-1,
  0, 4, 2,-1,-1,
  1, 5,-1,-1,-1,
  0, 4, 6,-1,-1,
  1, 3, 5, 7,-1,
  2, 4, 8,-1,-1,
  3, 7,-1,-1,-1,
  4, 6, 8,-1,-1,
  5, 7,-1,-1,-1,
};

/* 初期状態 */
char init_state[SIZE] = {
  1, 2, 3, 4, 5, 6, 7, 8, 0
};
```

マクロ定義の定数SIZEとMAX_STATEを変更します。あとは，初期状態とキューの初期化を変更します。空白の位置を示す配列space_postionの初期値の修正をお忘れなく。修正はこれだけです。詳細はソースファイルeight1.cを参照してください。実行結果は図10のようになりました。

手数は31手で局面は2つありました。実行時間は約13秒，結構時間がかかっていますね。そこで，もうひとつ工夫をしてみます。8パズルの場合，同じ駒を続けて動かすと，駒を元の場所に戻すことになってしまいます。これは元の局面に戻ることなので，わざわざ検索する必要はありません。いまのプログラムでは，このチェックを行っていないため無駄な検索が行われています。同じ駒を続けて動かさないようにすればもう少し速くなるでしょう。

8	6	7
2	5	4
3		1

6	4	7
8	5	
3	2	1

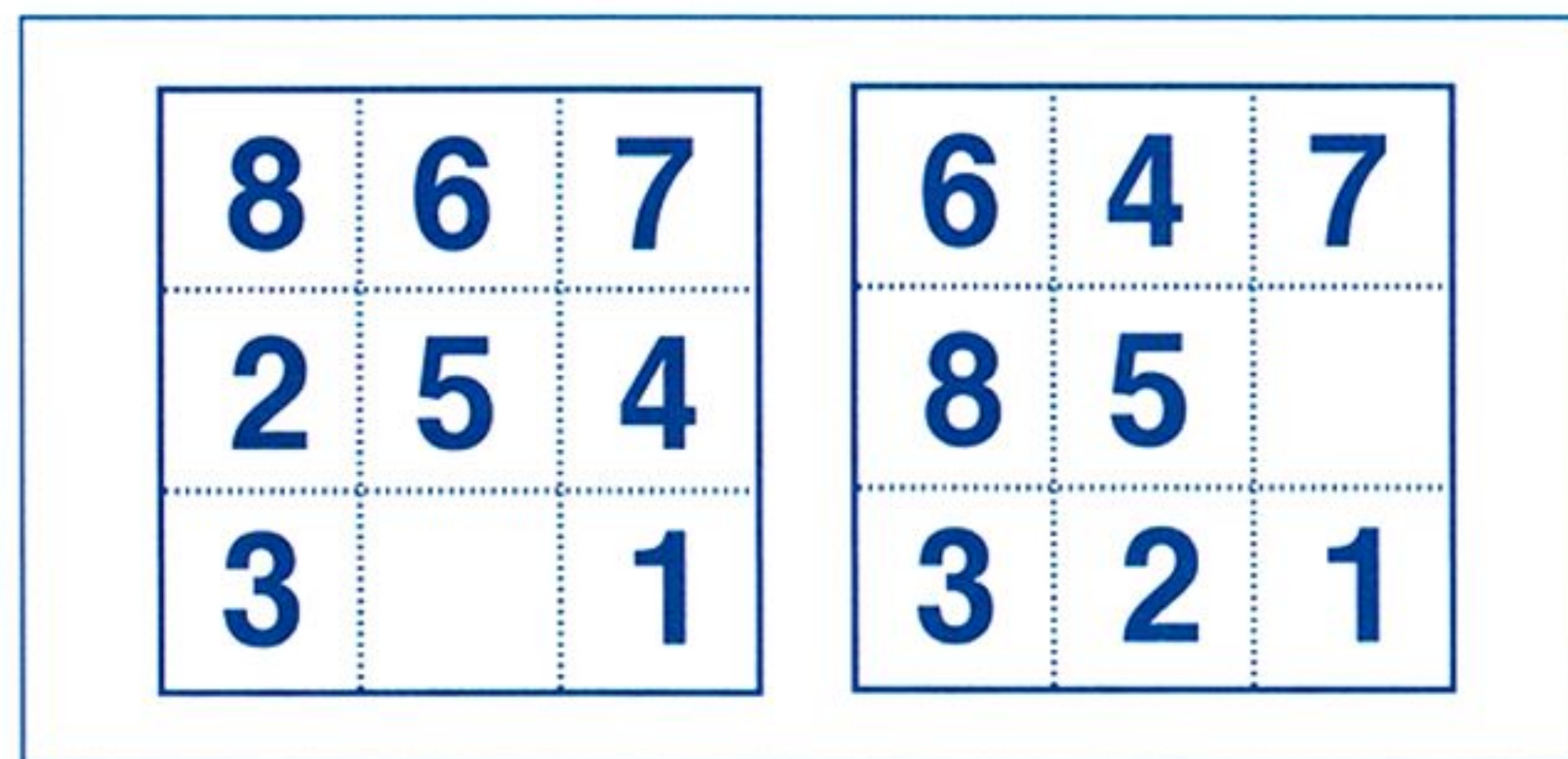



図10　31手で解ける局面

プログラムの改造は簡単です。動かした駒の種類を配列move_pieceに格納します。駒を動かすときは，move_pieceと違う種類の駒を動かすようにすればいいわけです。実際に改造を行ったプログラムがeight2.cです。実行結果は約8.5秒と速くなりましたが，それでも時間がかかりますね。

二分探索木でデータを探す場合，最下層のデータを見つける場合が最悪で，木の高さと同じ回数だけ比較が行われます。したがって，二分探索木はできる限り木の高さを低くするように構成したほうが，探索効率はよくなります。木の高さは，データ数をnとすると，データがランダムに挿入されれば，log n程度に収まります。ところが，ソートされたデータを二分探索木に挿入していくと，データは右側の木にしか挿入されず，連結リストと同じく線形探索になってしまいます。このプログラムではデータの総数が181440個なので，木の高さが18程度に収まっていれば最高の性能を発揮するのですが，実際に木の高さを求めると73という結果になりました。大きくバランスが崩れていることがわかります。

このように，二分探索木はバランスが崩れると性能が劣化する欠点があるのです。これを補うために，木のバランスを一定の範囲に収める「平衡木」が考案されています。有名なところでは，AVL木，2－3木，B木，B,木などがあります。これらの平衡木は，原理そのものは簡潔明瞭なのですが，実際のプログラムはとてもたいへんで，C言語初心者には荷が重すぎます。そこで，高速検索アルゴリズムの本命である「ハッシュ法」を使うことにします。

ハッシュ法

ハッシュ法は，コンパイラやインタプリタなどで予約語や関数名，変数名などの管理に使われている方法です。また，TclやPerlなど連想配列をサポートしているスクリプト言語がありますが，その実装にはハッシュ法が使われています。Perlで連想配列をハッシュと呼ぶのは，アルゴリズムの名称からきているのです。

ハッシュ法は，設計をうまく行えば1回の比較でデータを見つけることができます。実際，コンパイラの予約語のように探索するデータが固定されている場合は，そのように設計することが可能です。不特定多数のデータが探索対象になる場合は，すべてのデータを1回の比較で見つけることはできませんが，数回程度の比較で見つけるように設計することは可能です。

では，具体的に説明しましょう。ハッシュ法は，ハッシュ表と呼ばれるデータを格納する配列と，データを数値に変換するハッシュ関数を用意します。たとえば，ハッシュ表の大きさをnとすると，ハッシュ関数はデータを0からn－1までの整数値に変換するように作ります。この値をハッシュ値と呼びます。ハッシュ値はハッシュ表の添字に対応し，この位置にデータを格納します。つまり，ハッシュ関数によってデータを格納する位置を決める探索方法がハッシュ法なのです。

簡単な例題として，文字列を識別する処理を考えてみましょう。データが文字列の場合，次のハッシュ関数がよく使われます。

```
/＊ 文字列のためのハッシュ関数 ＊/
int hash（char ＊string）
{
  int value = 0;
  while（＊string != '¥0'）{
    value += ＊string++;
  }
  return value % HASH_SIZE;
}
```

HASH_SIZEはハッシュ表の大きさを定義したマクロです。valueをHASH_SIZEで除算した余りをハッシュ値としていることに注意してください。これでハッシュ値をハッシュ表の大きさに収めることができます。たとえば，HASH_SIZEを100とした場合，曜日を表す単語のハッシュ値は表2のようになります。

表2　曜日を表す文字列のハッシュ値

Sunday	28	Sun	10
Monday	16	Mon	98
Tuesday	35	Tue	2
Wednesday	32	Wed	88
Thursday	52	Thu	5
Friday	7	Fri	89
Saturday	45	Sat	96

単語のハッシュ値はすべて異なっていますね。曜日を識別するだけの処理であれば，これらの文字列をハッシュ表に登録すればいいのです。文字列のハッシュ値を計算し，そこに登録されているデータと比較します。等しいデータであれば，曜日を表す文字列であることがわかります。異なるデータであれば，その文字列は曜日を表していません。また，ハッシュ表に文字列が登録されていなければ，それも曜日を表していないことがわかります。このように，ハッシュ法を使えば1回の比較でデータを見つけることができます。

ただし，これはデータが限定されている場合の話です。たとえば，単語の出現回数をカウントする処理を考えてみましょう。テキストファイルのなかに含まれる単語の種類は，テキストを読んでみないとわかりません。したがって，テキストから単語を切り出し，それをハッシュ表に登録する処理が必要になります。ハッシュ表には有限の大きさしか割り当てることができないので，単語の種類がそれより多いと，ハッシュ値が重なる場合が必ず発生します。また，ハッシュ表が十分に大きくても，不特定多数のデータに対し，すべて異なるハッシュ値を生成するハッシュ関数を作ることは不可能です。つまり，異なったデータに対し，同じハッシュ値が生成される場合があるのです。これをハッシュ値の衝突といいます。データをハッシュ表に登録しようとしても，すでに先客が居座っているわけです。この場合，2種類の解決方法があります。

ひとつは空いている場所を探して，そこに入れる方法です。新しい場所を探すといっても，テーブルの先頭から線形探索するのではなく，最初とは違うハッシュ関数を用意して，新しくハッシュ値を計算させて場所を決めます。これを空いている場所が見つかるまで繰り返します。この方法だと，データの最大数はハッシュ表の大きさに制限されます。

もうひとつは，ハッシュ表に複数のデータを格納することです。配列にはひとつのデータしか格納できないので，複数個のデータをまとめて格納しておく工夫が必要になります。このときによく利用されるデータ構造が「連結リスト」です。ハッシュ表にはデータをそのまま格納しないで，連結リストへのポインタを格納すればいいのです。ハッシュ表からデータを探索する場合，まずハッシュ値を求め，そこに格納されている連結リストのなかからデータを探索します。

ただし，ハッシュ値の衝突が頻繁に発生すると，データを格納する連結リストが長くなるため，探索時間が余分にかかってしまいます。効率よく探索を行うためには，ハッシュ表の大きさとハッシュ関数の選択が重要になります。また，連結リストの代わりに二分探索木を使ってもかまいません。今回は十分に大きなハッシュ表を用意できるので，連結リストを使って複数のデータを格納することにします。

ちょっと寄り道

参考文献[2][4]には，順列を１対１に対応する数値に変換するアルゴリズムが紹介されています。つまり，ハッシュ値の衝突が起きないハッシュ関数とみなすことができます。この方法を使うことで，同一局面のチェックを高速に行うことができます。まあ，いつも都合のいいハッシュ関数が用意できるわけではありません。汎用的な方法を覚えておいたほうがいろいろなプログラムに応用することができるでしょう。

8パズル解法の高速化

それでは，ハッシュ法を使って８パズルの解法高速化に挑戦しましょう。このプログラムでも，連結リストを配列で表すことにします。セルとキューの定義は次のようになります。

```
/* 連結リスト */
typedef struct {
  char board[SIZE];
  int  next;
} CELL;

/* キュー */
CELL state[MAX_STATE + 1];     /* +1 はワーク領域 */
char space_postion[MAX_STATE];
char move_piece[MAX_STATE];
char move[MAX_STATE];
```

二分探索木と同様に，セルの連結はポインタの代わりに配列の添字で表すため，nextはintで定義します。終端はマクロNIL（－1）で表します。次にハッシュ表を定義します。

```
/* ハッシュ表 */
#define HASH_SIZE 19997
int hash_table[HASH_SIZE];
```

ハッシュ表もセルを指し示すのでintで定義します。大きさはHASH_SIZEで表します。参考文献[2]によると「この値が素数だと安心である」とのことなので，連結リストの長さが10より小さくなるように19997としました。ハッシュ関数も簡単です。

```
/* ハッシュ関数 */
int hash_value (int n)
{
  int i, value = 0;
  for (i = 0; i < SIZE; i++) {
    value = value * 10 + state[n].board[i];
  }
  return value % HASH_SIZE;
}
```

局面boardを10進数の数字とみなし，それをHASH_SIZEで割った余りをハッシュ値としています。最後にデータを登録する関数insert_hashを作ります。

```
/* ハッシュ表への登録 */
int insert_hash (int i)
{
  int h = hash_value (i);
  int n = hash_table[h];
  while (n != NIL) {
    if (memcmp (state[i].board, state[n].board, SIZE) == 0) {
      return FALSE;
    }
    n = state[n].next;
  }
  state[i].next = hash_table[h];
  hash_table[h] = i;
  return TRUE;
}
```

最初にハッシュ値を求め，ハッシュ表に格納されている連結リストを線形検索します。見つからない場合は，連結リストの先頭に登録します。とても簡単ですね。

あとは，プログラムを改造するだけです。関数searchでは，初期状態をハッシュ表に登録するためinsert_hashを呼び出し，局面をチェックする処理をinsert_hashに変更します。最後に関数mainでハッシュ表をNILに初期化する処理を追加します。これでプログラム(eight3.c)は完成です。

実際に実行してみると，時間は約2.5秒で約３倍程度の高速化となりました。ところで，ハッシュ法はハッシュ関数によって，性能が大きく左右されます。HASH_SIZEの選び方ひとつでも，実行時間は大きく異なります。たとえば，データ数のちょうど1/10である18144とすると，実行時間は約8.5秒と遅くなります。それよりも少ない16384では，逆に約2.5秒と遅くはなりません。このように，ハッシュ法では適切なハッシュ関数を用意するのが結構たいへんなのです。

ハッシュ法はデータを高速に検索できる優れたアルゴリズムです。データを検索するだけならば，二分探索木よりもハッシュ法が優れています。ですが，二分探索木にはハッシュ法にはない長所があります。二分探索木はデータの大小関係で構成されているので，左の木をたどることで最小値を，右の木をたどることで最大値を簡単に求めることができます。ハッシュ法で最大値や最小値を求めるには，すべてのデータを調べなければいけません。また，二分探索木では通りがけ順でデータを出力すれば，ソートされた結果を得ることができます。データの大小関係を処理する場合は，ハッシュ法よりも二分探索木を選ぶといいでしょう。

まとめ

パズルを例題にして，３回にわたり基本的なアルゴリズムとデータ構造を説明しました。解を探索するアルゴリズムではバックトラックと幅優先探索が基本です。データを高速に検索するには，二分探索木やハッシュ法を検討してみるといいでしょう。このほかにも，スタック，キュー，連結リスト，木構造，グラフなど基本的なデータ構造を使いました。これらのアルゴリズムやデータ構造は，パズルを解くだけではなく，ほかのプログラムを作るときにも必ず役に立ちます。ぜひ，活用してみてください。

次回は，もう少し複雑なパズルに挑戦してみましょう。お楽しみに。

参考文献

[1]A.V.Aho，J.E.Hopcroft，J.D.Ullman 「データ構造とアルゴリズム」培風館 1987
[2]奥村晴彦 「C言語による最新アルゴリズム事典」技術評論社 1991
[3]近藤嘉雪 「Cプログラマのためのアルゴリズムとデータ構造」 ソフトバンク 1998
[4]三木太郎 「特集コンピュータパズルへの招待 スライディングブロック編」C magazine 1996年2月号 ソフトバンク
[5]広井誠 「続・サルでも書けるCプログラム講座第12回」月刊・電脳倶楽部 Vol.95 満開製作所

Computer Architecture

コンピュータアーキテクチャ
その直感的アプローチ③

パイプライン処理の概念と実際

中森 章 *Nakamori Akira*

今回はパイプラインについて説明する。パイプラインとはMPUの命令実行を高速化する手法のひとつであり，現在では，ほとんどすべてのMPUで採用されている。本稿では，一度に1命令を実行する通常のパイプライン処理について解説する。このパイプラインは，2命令以上を同時実行するスーパースカラに対応して，特にシングルパイプラインと呼ばれることもある。ユニスカラパイプライン，あるいは単にスカラパイプラインとも呼ばれることがあるようである。

●パイプラインとは

コンピュータの性能を向上させる方法についてはいろいろなものが考案されている。パイプラインとはハードウェアを並列化して性能を向上させるための一般的な手法である。その基本的な考え方は，プログラム内蔵方式を提唱したフォン・ノイマンによってすでに提案されていたという。たとえば，MPUの命令実行に比べて10倍以上も遅いメモリアクセスが存在する状況下で効率的に命令の処理を行うために，命令の実行とメモリアクセスをオーバーラップして処理することが考えられた。これが，パイプライン処理の原型である。

パイプラインの基本的な考え方はごく自然なものである。なにもコンピュータの技術に固有なものではない。電子部品工場などで行われている流れ作業はパイプラインそのものである。実際，パイプラインの呼び名は，石油が次々とパイプを通過していく石油化学パイプラインと動作が似ていることに由来している。

各工程が1単位時間かかるN工程からなる処理を考える。単純に考えるとこの処理を終了するためにはN時間を要する（図1(a)）。これをN人の人が流れ作業によって各工程を分担し，前の工程から受け継いだ製品に1単位時間で加工を施してあとの工程に引き継ぐようにする（図1(b)）。この場合，元々の処理ではN時間にひとつしか製品が完成しないが，流れ作業では見かけ上，1単位時間にひとつの製品が完成することになる。つまり，処理速度はN倍に改善される。これがパイプラインの原理である。また，各工程をパイプラインのステージという。段という表現も使われ，N工程から構成されるパイプラインはNステージパイプライン，またはN段パイプラインと呼ばれる。

ところで，あるステージを分担する人が手間取ってそこでの処理を1単位時間以内に終わらせることができないような場合はパイプライン処理に乱れが生じ処理性能が低下する。パイプラインステージでの処理を単位時間内に終わらせることを阻害する要因をハザードという。

パイプライン処理をコンピュータに適用する場合は各ステージが並列に処理できることが前提である。ハードウェア資源を共有するステージがあると，ハザードが生じ，待ち合わせが必要になる。逆にいうと，ハードウェア資源が競合しないようにパイプラインステージを分割することが設計者の腕の見せどころである。

パイプライン処理はまず大型計算機で採用された。その後，半導体の集積技術が進み，MPUでも大量のトランジスタが利用可能になると，MPUにも採用されるようになる。パイプライン処理の採用を大々的に表明したMPUはNECのV60のような気がする。それ以前のインテルの8086でもオペランドフェッチと実行をパイプライン化していたが，インテルがパイプラインを明言したのは80386以後と記憶している（編注：最近のインテルの発言ではPentium以降となっている）。一方，68000系のMPUも古くからバスサイクル同期のパイプライン処理をしていたきらいがある。しかし，こちらもパイプラインを明言したのは68060で初めてのような気がする。68060はすでにスーパースカラ構造になっていたので，シングルパイプライン時代の68000系のパイプライン構造は不明である。

●CISCのパイプライン

(1)パイプラインステージ

のっけからこういうのも恐縮だが，CISCのパイプラインはどうも明確ではない。CISC型のMPUのパイプラインの説明でよく使われる表現は，いくつかの機能ブロックが同時に動作するというものである。各機能ブロックがパイプラインの各ステージに対応し，それらが絶えず独立して動き続けているというイメージである。これは現在のクロックに同期してステージが順次進行していくイメージとは少々異なる。命令フェッチ，命令デコード，実行という流れは一応あるのだが，実際には各機能ユニットは要求（開始）信号と許可（終了）信号によるハンドシェーク処理で実現されていることが多い（クロック同期という感じが薄い）。その意味でパイプラインのステージ数は機能ブロックの数に等しい。ただし，命令の種類によってはすべての機能ブロックを使用するとは限らないので，実際のステージ数は実行時に動的に変化する。しかし，そこで行われているパイプライン処理は典型的なステージの移行で説明されていることが多く，実装とイメージのギャップを感じる。しかし，ここでも一応パイプラインのイメージを重視して説明する。

CISC型のMPUの典型的な命令はレジスタとメモリ間の演算である。たとえば，

```
ADD R1, MEM  // レジスタR1とメモリ
                MEMの内容を加算し
                R1に格納する
ADD MEM, R1  // メモリMEMとレジス
                タR1の内容を加算し
                MEMに格納する
```

などが考えられる。これらの命令処理の流れはだいたい次のようになっている。なお，パイプラインを進めるきっかけとなるのは外部から入力されるクロックであり，通常はクロックの1～3サイクルをパイプラインステージの単位時間としている。

1. 命令フェッチ（IF）：PC（プログラムカウンタ）の指すアドレスのメモリから命令を取り込む
2. 命令デコード（ID）：命令をデコードして命令処理に必要な情報を取り出す
3. アドレス生成（EA）：デコード結果を基に命令が指定する実効アドレスを計算する
4. アドレス変換（AT）：実効アドレス（仮想

アドレス）を論理アドレスに変換する
5. オペランドフェッチ（MEM）：論理アドレスで指定されるメモリまたはレジスタからデータをリードする
6. 命令実行（EX）：デコード結果，フェッチしたオペランドを基に命令を処理する
7. オペランド格納（WB）：論理アドレスで指定されるメモリまたはレジスタに命令実行での処理結果を格納する

[1.] PCの更新（IF）：新たなPCを計算して，そのアドレスのメモリから命令を取り出す

⋮

命令フェッチのためのアドレス変換がないと訝る人のために付け加えておくと，命令フェッチは正確にはPCではなく，それをアドレス変換した結果を保持しているプリフェッチポインタの値によって行われる。命令フェッチは，一度アドレス変換をしたら，MMUが管理するページ境界を超えるか分岐を生じるまでアドレス変換をする必要がない（変換しても同じページを指すのは明らかなので）。ページ超えによる変換要求が生じる場合はパイプラインが停止するし，分岐の場合は分岐先がオペランドになるのでオペランドのアドレス変換に含まれる。このため，一連のパイプライン処理において命令フェッチのアドレス変換を考える必要はない。

もっとも，こういった場合分けをするより，命令フェッチごとにアドレス変換をしたほうが制御が単純化されるという考えもある。RISC型のMPUではこの方式（毎回変換）をとることが多く，命令フェッチのアドレス変換もパイプラインのステージに組み込まれている。

話が横に逸れたが，CISCのパイプラインは上記の7ステージに分けることができる。これらのステージがオーバーラップして命令を処理する（図2）。しかし，実際の実装ではこんなに整然と細分化されていることは稀である。命令の種類によってはいくつかのステージ（たとえばメモリアクセス）がない。また，いくつかのステージを同時に処理したり，いくつかのステージをあわせてひとつのステージとする。

複数のステージの同時実行は一見効果的であるが，パイプラインが前のステージの結果を利用して進むことを考えると無意味である。複数のステージを合併することは，ややもすれば，ステージ間の処理時間を不均一にすることになり，パイプライン処理に乱れを生じやすい。当然パイプラインのスループット（単位時間に完了する命令数）は低下する。初期のMPUのパイプライン処理は有効に機能していない場合が多いように思う。というか，大型計算機で採用されてきたパイプライン処理をMPUでいかに実現すべきか試行錯誤の時代だったのかもしれない。

(2)パイプラインハザード

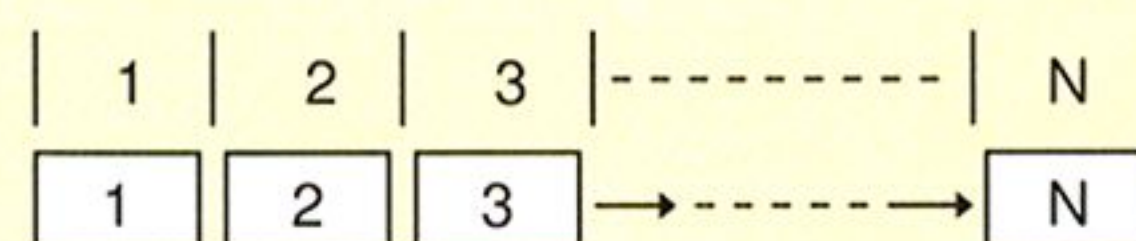
また，パイプラインが十分に機能していても構造上避けられないハザードが存在する。たとえば，メモリ内のオペランドをフェッチする場合，リードが完了するまでパイプラインが停止（ストール）する（図3(a)）。後続命令が直前の命令で変更されるレジスタの値を使用する場合はレジスタの値が確定するまで待ち合わせが必要である（図3(b)）。後続命令が直前の命令で変更されるレジスタの値を使用してアドレス計算を行う場合も同様である（図3(c)）。このほかにもハザードはあるが，詳しくはRISCのパイプラインの項で説明する。

図3の(b)，(c)のケースは，命令の順序をコンパイラなどが適当に並び替えることでハザードを回避することができる。CISCの場合，ハザードの解消はもっぱらコンパイラに頼っていた（ハザードを回避しない場合はストールするだけで，誤動作するわけではない）。RISCの場合は，コンパイラはもちろんであるが，MPU自身の構造としてハザードを解消する仕組みを持たせようとしている。

(3)ステージの処理時間が不均一なパイプライン

さて，パイプラインのステージ間の実行時間が均一でない場合を考える。実際問題として，

図1　パイプライン処理

(a) N工程からなる処理

(b) パイプラインによる処理

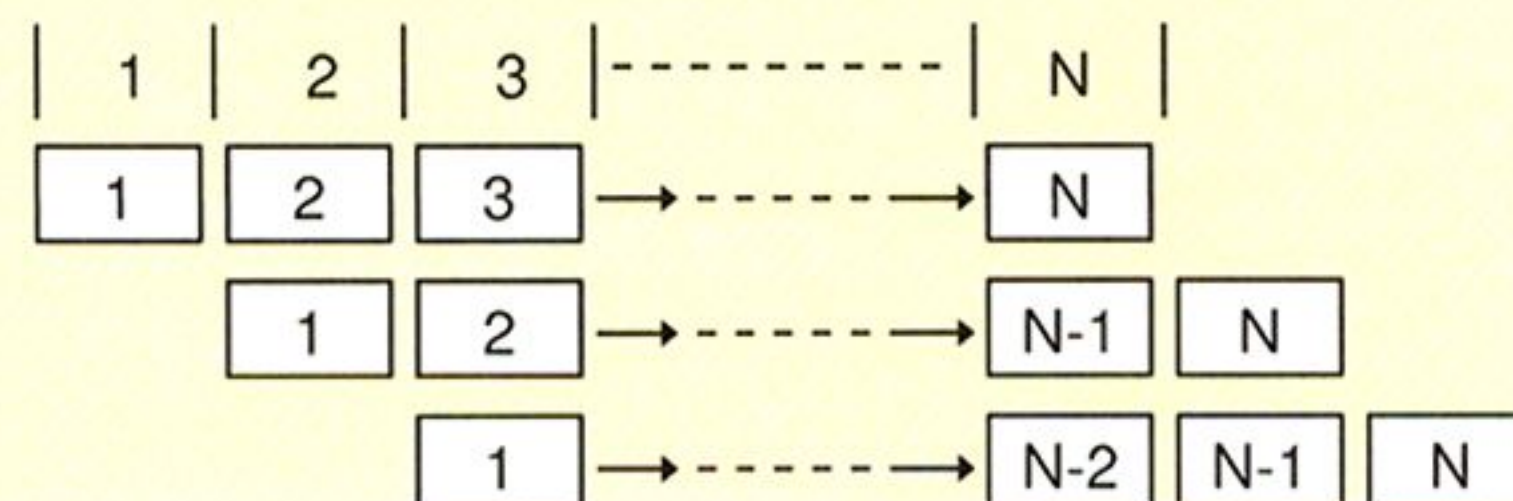



図2　パイプライン処理の流れ

(a) ステージと機能ブロックの関係

P C：プログラムカウンタ
I R：命令レジスタ
TLB：Translation Lookaside Buffer
OPR：オペランド リード レジスタ
OPW：オペランド ライト レジスタ

(b) スムーズなパイプラインの流れ

	1	2	3	4	5	6	7	8	9	10
命令1	IF	ID	EA	AT	MEM	EX	WB			
命令2		IF	ID	EA	AT	MEM	EX	WB		
命令3			IF	ID	EA	AT	MEM	EX	WB	
命令4				IF	ID	EA	AT	MEM	EX	WB

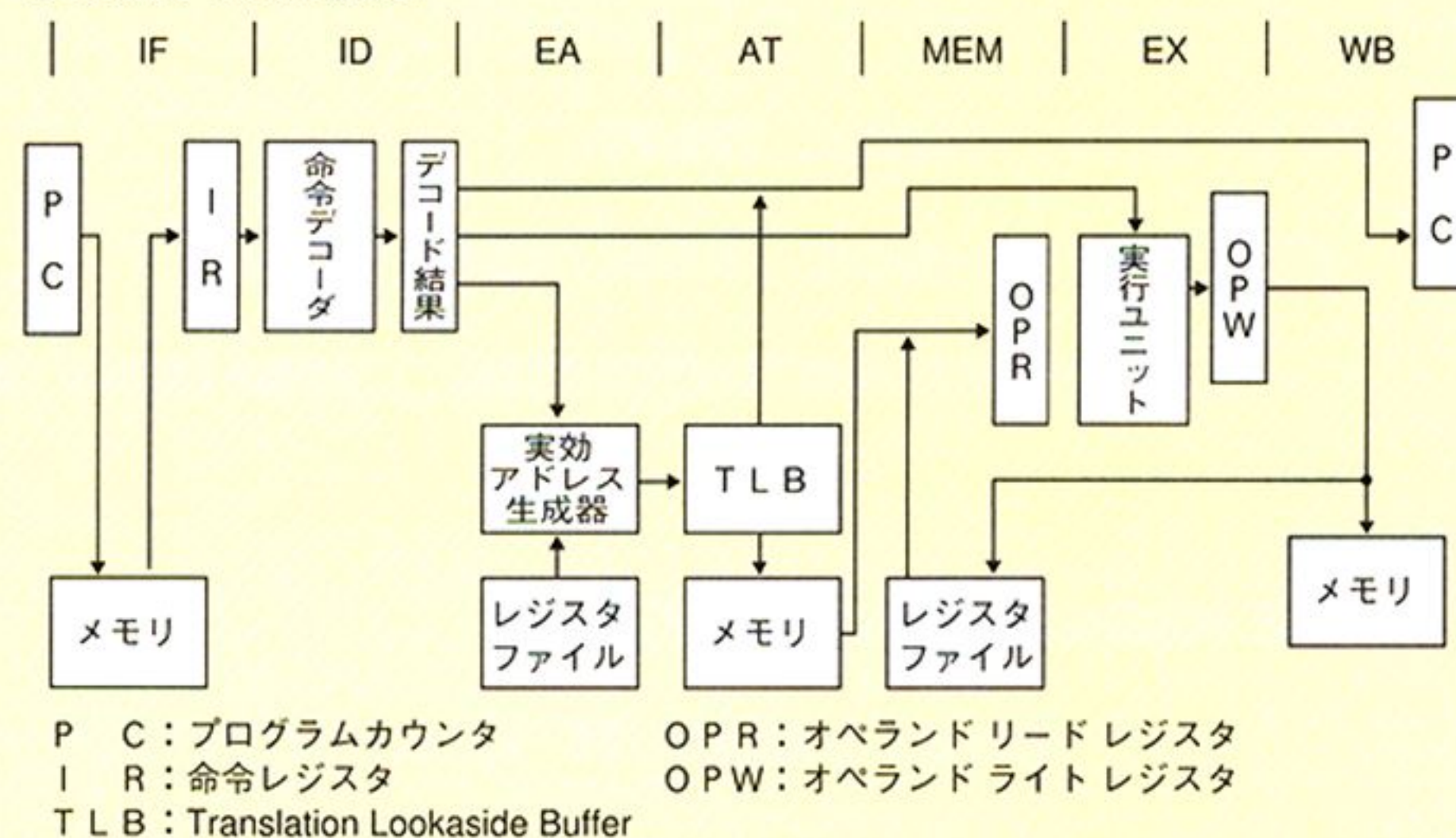
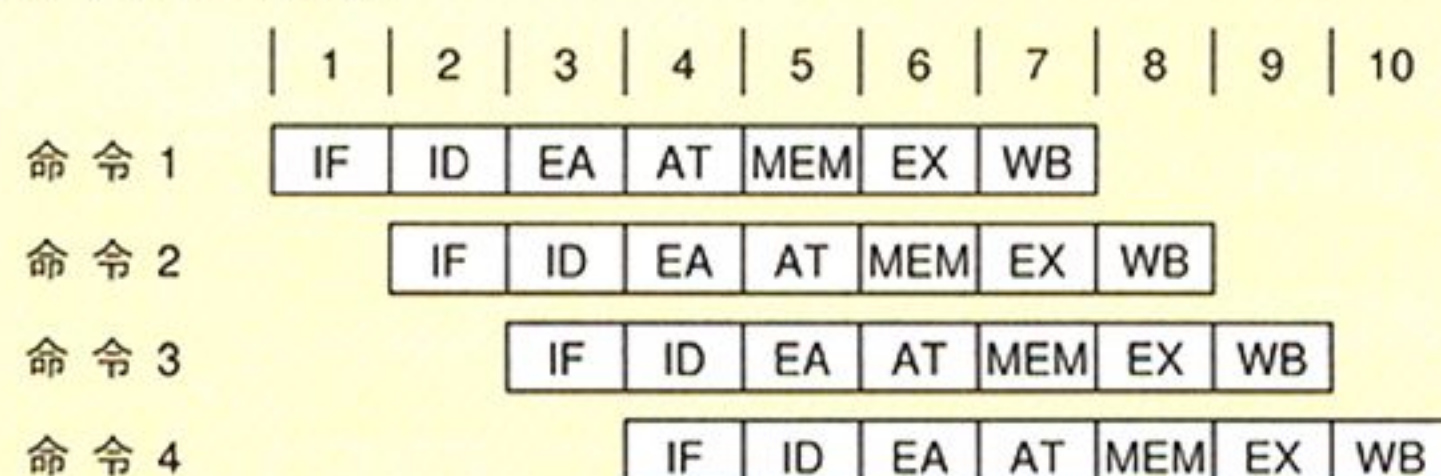



図3 パイプラインのストール

(a) メモリアクセスによるストール

	1	2	3	4	5	6	7	8	9	10	11	12
命令1 (メモリアクセス)	IF	ID	EA	AT	MEM	MEM	MEM	EX	WB			
命令2		IF	ID	EA	AT	AT	AT	MEM	EX	WB		
命令3			IF	ID	EA	EA	EA	AT	MEM	EX	WB	
命令4				IF	ID	ID	ID	EA	AT	MEM	EX	WB

ストール

(b) レジスタ依存によるハザード

	1	2	3	4	5	6	7	8	9	10	11	12
R1←R1+R2	IF	ID	EA	AT	MEM	EX	WB					
R3←R3+R1		IF	ID	EA	AT	MEM	MEM	EX	WB			
NEXT			IF	ID	EA	AT	AT	MEM	EX	WB		

ストール

(c) アドレス計算によるハザード

	1	2	3	4	5	6	7	8	9	10	11	12
R1←R1+R2	IF	ID	EA	AT	MEM	EX	WB					
R3←R3+(R1)		IF	ID	EA	EA	EA	EA	AT	MEM	EX	WB	
NEXT			IF	ID	ID	ID	ID	EA	AT	MEM	EX	WB

ストール

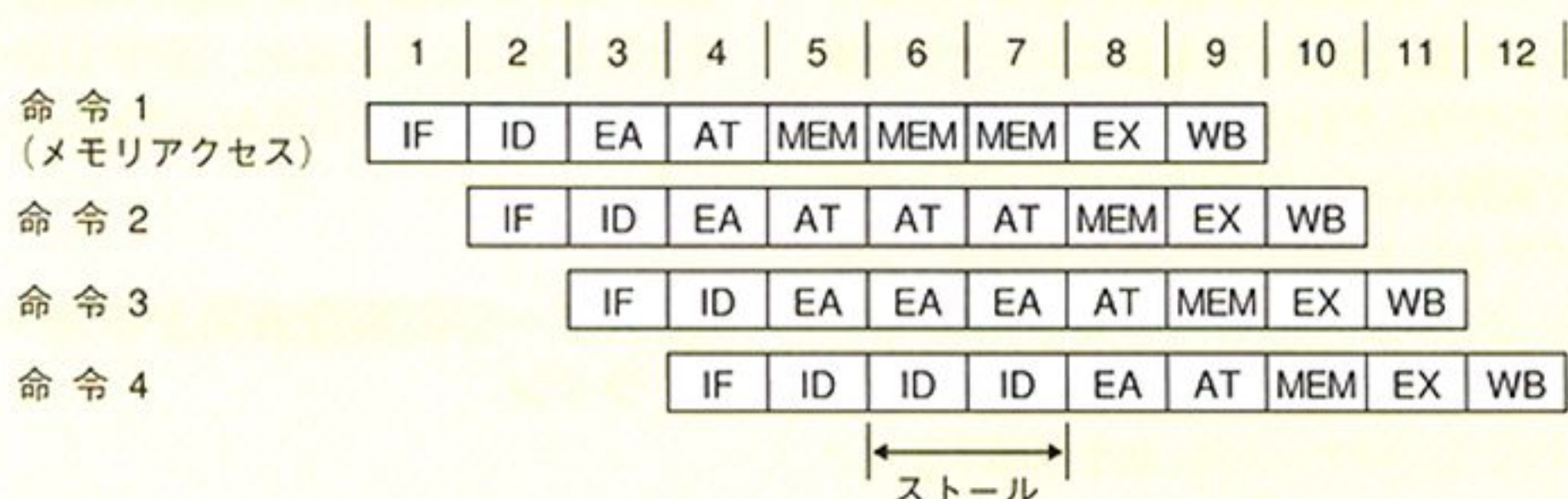
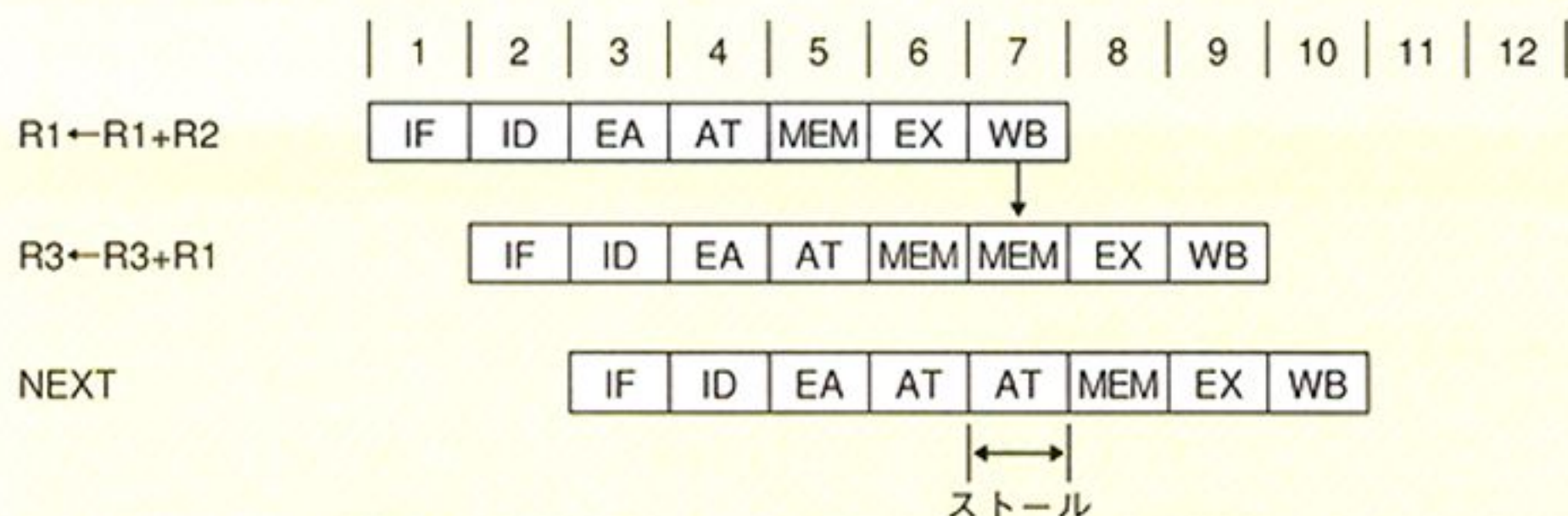
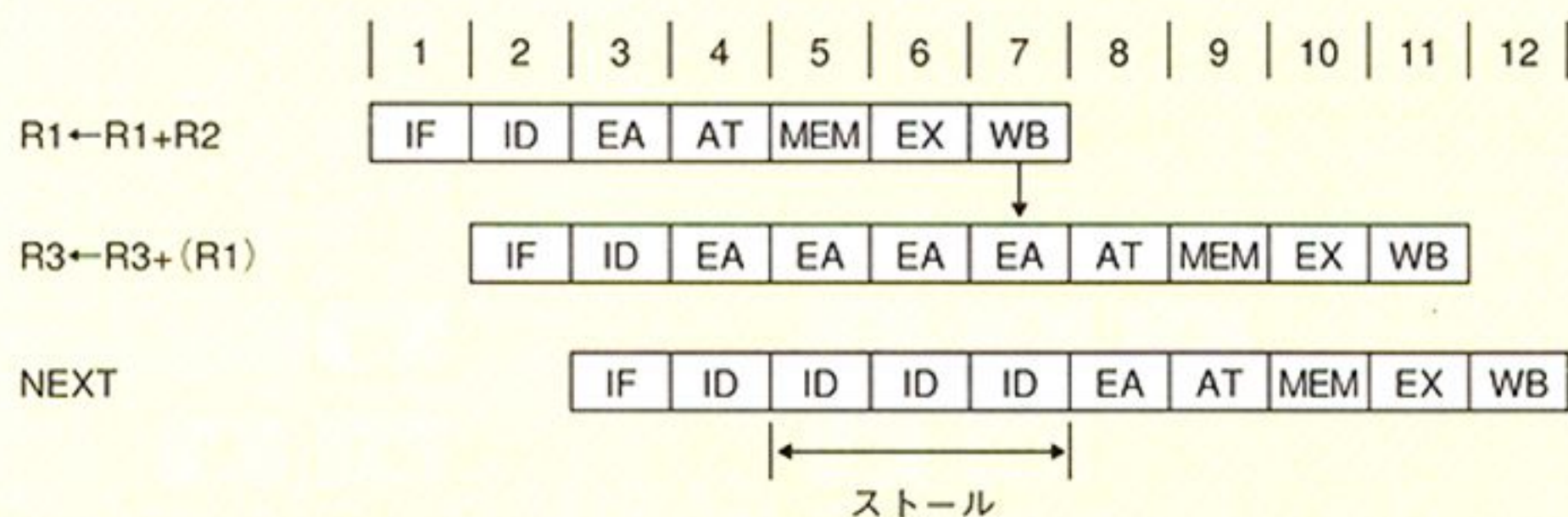



図4 パイプラインステージが不均一なパイプライン

(a) 実行ステージ(EX)が長いパイプライン

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15

命令1: IF | ID | EA | AT | MEM | EX | WB

命令2: IF | ID | EA | AT | MEM | EX | WB

命令3: IF | ID | EA | AT | MEM | EX | WB

(b) 命令フェッチステージ(IF)で性能が決まる

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15

命令1: IF | ID | EA | AT | MEM | EX | WB

命令2: IF | ID | EA | AT | MEM | EX | WB

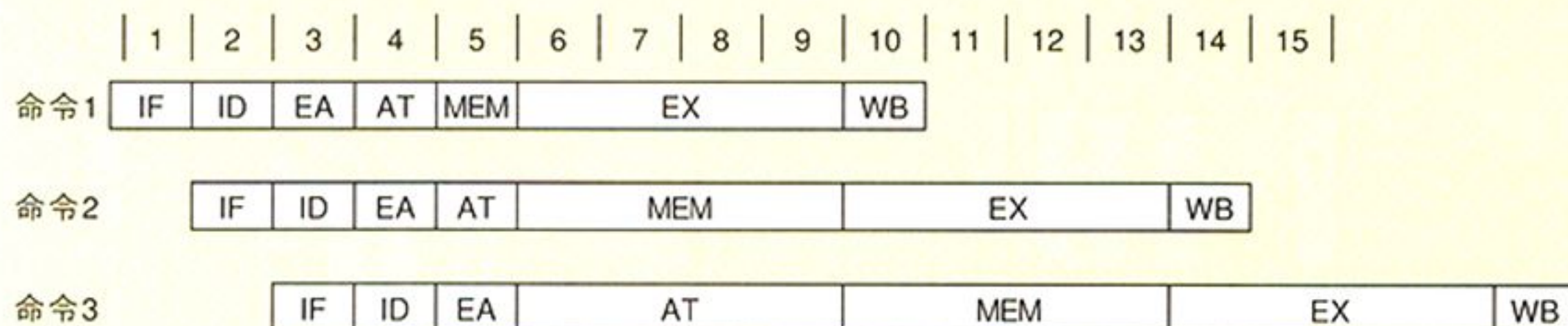
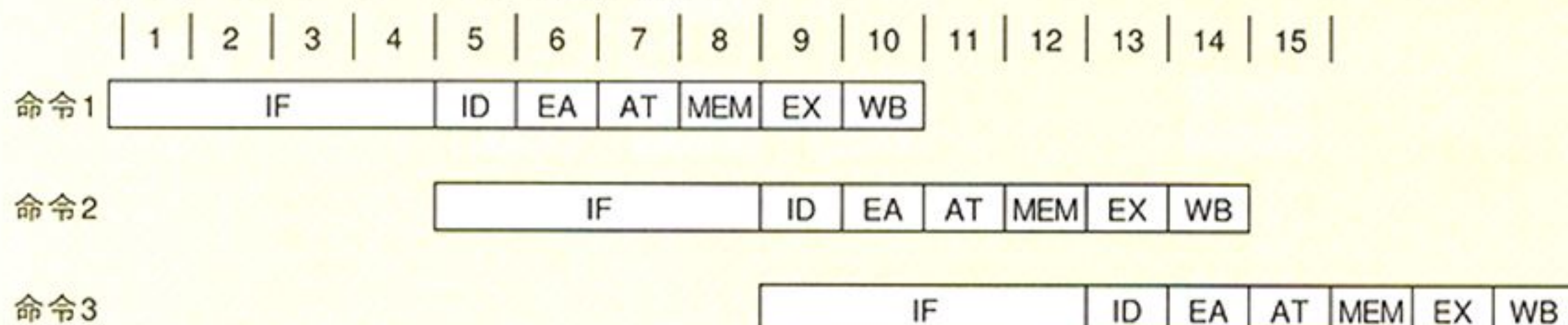
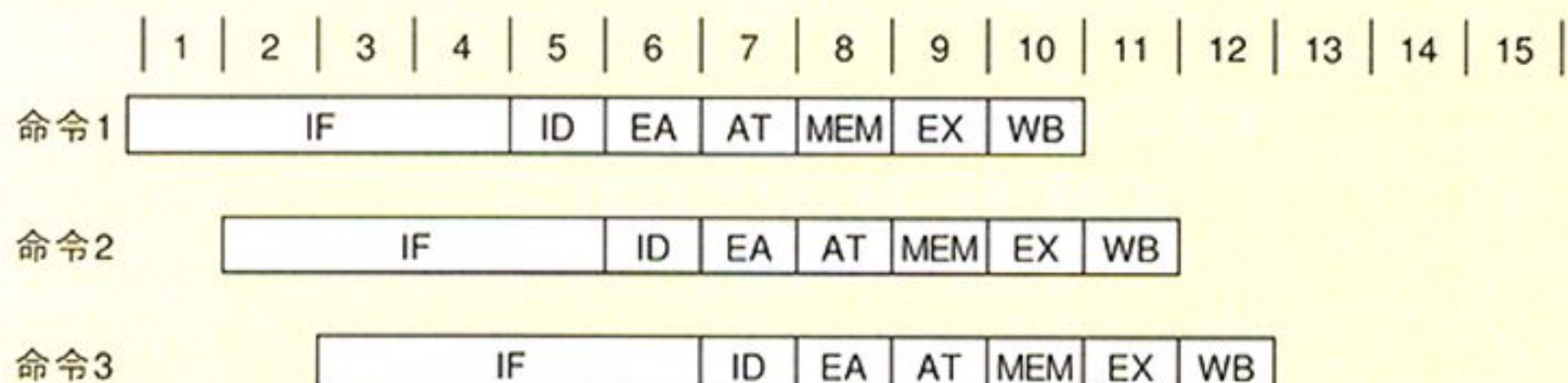
命令3: IF | ID | EA | AT | MEM | EX | WB

(c) プリフェッチを行うパイプライン

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15

命令1: IF | ID | EA | AT | MEM | EX | WB

命令2: IF | ID | EA | AT | MEM | EX | WB

命令3: IF | ID | EA | AT | MEM | EX | WB

CISCは複雑な処理を1命令で行うため，実行ステージはほかのステージに比べて実行時間が長い。いま，実行ステージ(EX)の処理が4単位時間かかるものとする(図4(a))。この場合，パイプラインのスループットは実行ステージの処理時間に依存する。ほかのステージの処理時間は実行ステージの処理時間に隠れてしまう。実行ステージの処理時間が長いだけならまだいい。ほかのステージもまちまちな処理時間を有する場合は悲惨である。不均一であればあるほど，パイプラインの処理時間は各パイプラインステージの処理時間の総和に近づいていく(パイプラインの意味がなくなる)。このため，実行ステージ以外のステージの処理時間を均一にすることが肝要である(マイクロコードで実行することが多いCSICで実行ステージの処理時間にばらつきがあるのはしかたないので)。

こうなると，いちばんネックになるのがメモリアクセスを伴う命令フェッチステージ(IF)である。CISCでは単純な命令もあり，そのEXステージの処理時間は短い場合もある。一方，命令フェッチなどのメモリアクセスは命令によって異なることはないので，平均すれば命令フェッチステージのほうが性能に与える影響が大きい。

先に述べたように，CISCのパイプラインはユニット(ステージ)間のハンドシェークが基本で，クロック同期でないことが多いので，図4(b)のように，処理性能が命令フェッチネックになることもある。このような状況下で，パイプライン処理をスムーズに行うためには，パイプライン処理の単位時間を(クロックではなく)バスサイクルと同一にすることも考えられる。しかし，この方式だと，バスサイクル自身が遅いので，パイプライン全体の処理速度が低下してしまう。そこで，命令実行の間隔を縫って命令をプリフェッチ(先取り)する方式もある。絶えず十分な量の命令をMPU内部に取り込み続けてプリフェッチキューと呼ばれるバッファに蓄えておく(一種の命令キャッシュ)ことで，命令フェッチステージを，見かけ上，最小時間で処理することができる(図4(c))。

命令フェッチの次に処理時間がかかるのは命令デコードである。x86アーキテクチャのように可変長命令の体系を採用するMPUにおいては，単位時間で命令をデコードするには命令のエンコードが複雑すぎる。この場合は命令デコードに2単位時間をかける。まず大雑把に位置合わせとプリデコードを行い，次に完全にデコードする。ただし，この処理はひとつのパイプラインステージではなく，2つのステージに分けて行う場合が多い。このためパイプラインのスループットに与える影響はない。もっとも，デコードに時間をかければ，分岐命令の処理は遅くなるが，これは後述のRISCのパイプラインの項で説明する。

余談であるが，命令実行と命令デコードは，プリフェッチとは異なり，処理時間を最小化する有効な手段はない。このため，命令の実行時間は実行ステージとデコードステージの処理時間が長いほうで規定される。

CISCのパイプライン処理においては，最後のライトバック（WB）ステージがないものもある。CISCの命令実行はマイクロコードで実現されることが多く，レジスタへのアクセスはオペランドフェッチ（MEM）ステージではなく，マイクロコードの処理が始まってから行う。つまり実行（EX）ステージ内でレジスタへのリード/ライトを行うため，わざわざライトバックステージは設けない。この方式を採用する場合，図3(b)に示すレジスタ依存によるハザードは存在しない。

歴史的に見れば，キャッシュメモリ（高速なローカルメモリ）がまだ高価で外付けのキャッシュすら現実的でなかった時代，プロセッサの処理はメモリからの命令フェッチにいちばん時間がかかっていた。プロセッサの性能を上げるために，当然の流れとして，フェッチする命令数を減らすこと，1命令で行う処理を増やすことが考えられた。結果として，上述したように実行ステージが長くなる傾向になるのだが，多くの場合はいちばん時間のかかる命令フェッチと，あまり時間のかからない命令のデコードおよび実行をオーバーラップ（パイプライン処理）させて実行効率を上げることが可能になる。これが，その時代の最適解であった。そして，それこそがCISCの考え方である。

その後RISCという選択肢が現れてきた背景には，キャッシュが一般的になり，命令フェッチがもはやプログラムの実行に支配的でなくなったことがある。命令のデコードや実行時間が命令フェッチ時間の陰に隠れなくなり，実行する命令数よりも1命令の実行時間のほうが性能面で支配的になった。RISCでは，基本的に1クロック実行なので，CISCに比べて命令実行時間が1/3から1/5になる。1命令が単純な分，同じ処理に要するコード量は増加するが，RISCになることによる命令数の増加はわずか30％から50％であるという。差し引き，性能は向上する。また，RISCでは命令が基本操作に限定されているのでコンパイラによる最適化が行いやすいという利点もある。まあ，現実には，基本的な命令だけで優れた最適化ができるということをMIPSやSPARCなどのコンパイラが実証したせいでRISCがメジャーになったともいえるのだが，CISCからRISCの流れは歴史の必然であった。

●RISCのパイプライン

(1)パイプラインステージ

CISCの初期においてパイプライン処理は有効に機能していたとはいえない。各MPUメーカーがパイプラインを強調しなかったのは，それが性能に寄与していなかったからではと考えられる。しかし，RISCの登場によってパイプライン処理はにわかに脚光を浴びる。RISCのパイプラインは，CISCとは異なり，全命令でパイプラインのステージ数は固定であることが多い。筆者だけの感覚かもしれないが，命令フェッチ，命令デコード，実行という処理の流れも，その区切りが明確になっているように感じる。

RISCの存在意義は，いかにパイプライン処理を効率的に実現できるかにかかっているといっても過言ではない。このため，RISCでは命令やオペランドをキャッシュからフェッチすることを前提としている。通常のメモリはアクセス時間が遅いので，メモリアクセスステージの処理時間がほかのステージに比べて長くなり，効率的なパイプライン処理を行うことはできない。ステージの処理時間を均一化するためキャッシュの導入は必然だったといえる。キャッシュの導入により，メモリアクセスステージが1または2クロックで処理できるようになる。

RISCのパイプラインはコンピュータアーキテクチャの超有名な教科書で学ぶことができる。それが，HennessyとPattersonによる『コンピュータアーキテクチャ』（通称ヘネパタ）である。この教科書では仮想的なMPUとしてDLXというMPUを定義し，そのパイプラインとして次の5ステージの処理を提案している。もっともDLXはMIPSのR2000/R3000と非常に近い（同じ？）構造をしており，以下はR3000のパイプラインそのものということもできる。ただし，ヘネパタではメモリがキャッシュであることを特に強調してはいない。

1. 命令フェッチ（IF）：命令キャッシュから命令を取り出す
2. 命令デコード（RF）：フェッチした命令をデコードする。同時にレジスタオペランドをフェッチする
3. 命令実行（EX）：デコード結果とフェッチしたレジスタの値を基に命令を実行する。ロード/ストア命令の場合は実効アドレスの計算を行う。分岐命令の場合は分岐先アドレスを計算する
4. オペランドフェッチ（MEM）：EXステージで計算したアドレスに対応するメモリの値をデータキャッシュよりリードする
5. ライトバック（WB）：EXステージで計算した結果，またはMEMステージでフェッチしたオペランドをレジスタに格納する。ストア命令の場合はデータキャッシュにライトする

上のパイプラインではアドレス変換のステージがないが，これはIFまたはMEMステージに先立って行われる。この詳しい説明は後述のR3000のパイプラインの実際の項に譲る。

RISCのパイプラインの特徴はアドレス計算をする専用のステージがなく，EXステージで代用している点である。このためアドレス計算用の演算器と命令実行用の演算器（実際は加算器）をそれぞれ別個に用意する必要はない。これはRISCの「ロード/ストアアーキテクチャ」という特徴に由来する。つまりひとつの命令では2回加算を行うことがない。つまり，1命令で1回だけ演算器を使用するという制限の下で，レジスタとレジスタ間の加算，または，アドレス計算（ロード/ストア命令）は別の命令に分かれて定義されている。RISCのパイプライン処理を図5に示す。

(2)データハザードとフォワーディング

さて，CISCのパイプラインの項で述べたハザードはRISCのパイプラインでも発生する。それを詳しく見ていこう。まずはレジスタの依存関係

図5 RISCのパイプライン

(a) ステージと機能ブロックの関係

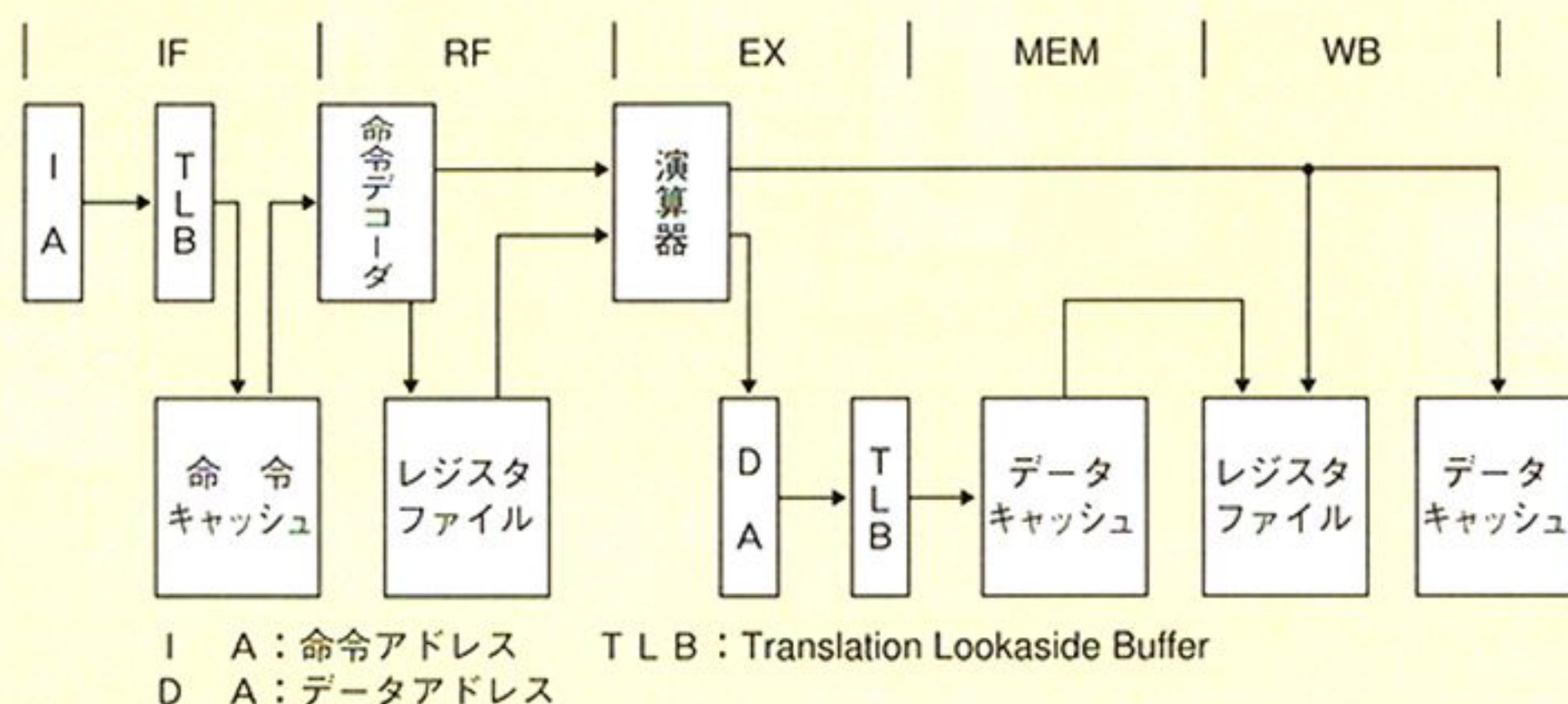
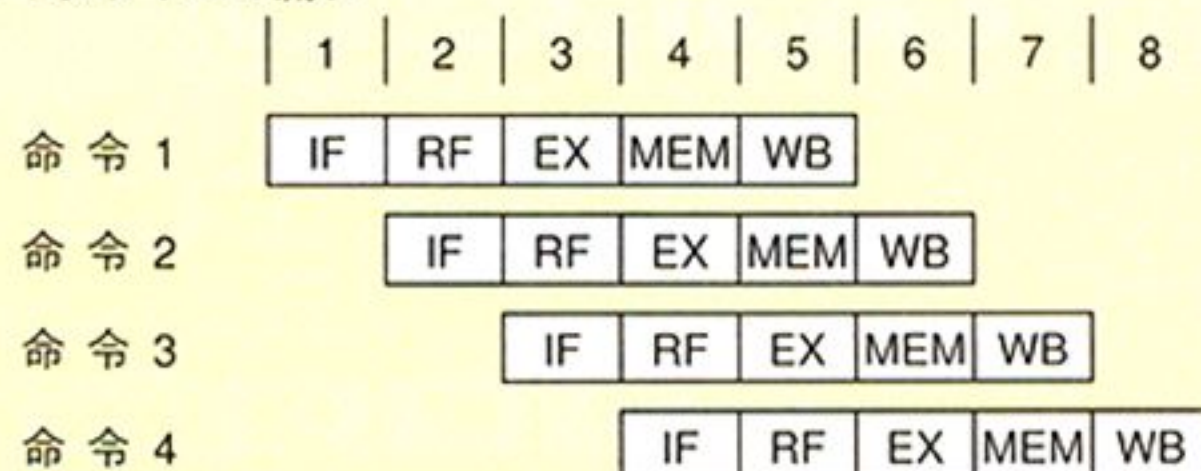
IF | RF | EX | MEM | WB

IA　TLB　命令デコーダ　演算器

命令キャッシュ　レジスタファイル　DA　TLB　データキャッシュ　レジスタファイル　データキャッシュ

I　A：命令アドレス　T L B：Translation Lookaside Buffer
D　A：データアドレス

(b) スムーズなパイプラインの流れ

	1	2	3	4	5	6	7	8
命令1	IF	RF	EX	MEM	WB			
命令2		IF	RF	EX	MEM	WB		
命令3			IF	RF	EX	MEM	WB	
命令4				IF	RF	EX	MEM	WB

に起因するハザードである。レジスタ間のリード/ライトの関係で次の4種類が考えられる。

(i) RAW (Read After Write) ハザード
　これは，レジスタライトの完了前に後続命令によって同一のレジスタをリードしようとした場合に生じる（図6）。

(ii) WAR (Write After Read) ハザード
　これは，レジスタから値をリードする前に後続命令によって同一のレジスタにライトをしようとした場合に生じる。

(iii) WAW (Write After Write) ハザード
　これは，同一レジスタへのライト順序が狂う場合に生じる。

(iv) RAR (Read After Read) ハザード
　一応挙げたが，レジスタへの変更が伴わないのでこのようなハザードは存在しない。

　以上はデータに起因するハザードなので総称してデータハザードと呼ばれる。しかし，(ii)，(iii)のハザードは命令の実行順序が狂わない限り発生しない。通常のパイプラインでは発生しないが，スーパースカラ構造では発生することがある。これは次回説明する。当面の課題は(i)のRAWハザードである。これは，フォワーディング，バイパス，または，ショートサーキットと呼ばれる手法で解決可能である。つまり，EX，MEM，WBステージからRFステージへのバイパス回路を設けることで解決できる（図7）。RISCでは，パイプライン処理を乱さないために，フォワーディングは半ば常識である。しかし，パイプラインのステージ数が多い場合，具体的には，レジスタをフェッチするステージ(RF)とレジスタへの書き込みステージ(WB)の間の段数が多いと，各ステージからRFステージへのバイパス経路がその段数分必要なので，実行ステージ(EX)へ与えるデータのセレクタが巨大になってしまう。もちろん動作周波数にも影響を与える。どの程度フォワーディングを行うかは悩ましいところである。

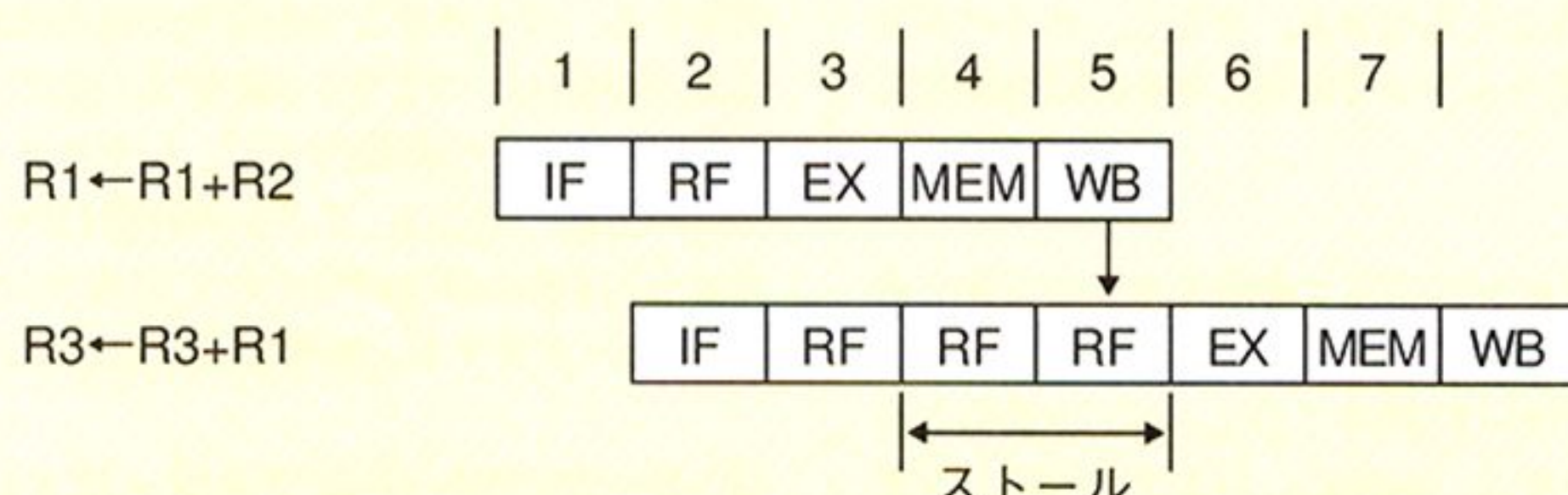
図6　RAWハザード

図7　バイパス回路とフォワーディング

(3)ロード遅延と遅延ロード

　ロードした値を直後の命令で使用する場合を考える。この場合，MEMステージで値が初めて確定する。このとき後続命令はEXステージにあるのでフォワーディングは不可能である（図8(a)）。なにも対処しないと変更前のレジスタの値をフェッチしてしまう。これをロード遅延という。このため，プログラムの意図どおりに命令を処理するには，パイプラインのインタロックが必要となる。インタロックとはハザードの有無をテストし，ハザードがある場合はハザード原因が解決するまでパイプラインを停止する機構である。また，停止しているサイクルをパイプラインストール（パイプラインバブル）と呼ぶ。図5で示す5ステージ構成のパイプラインなら1クロックストールさせればよい（図8(b)）。

　パイプラインのストールは処理性能の低下を意味する。それを回避する手法のひとつは命令の順序を入れ替えることである。いまの場合，1クロック分(1命令分)待ち合わせればいいので，ロードした値を参照する命令と後続の無関係な命令を入れ替えればよい。入れ替えるべき適当な命令がない場合はNOP命令を挿入することになる（図8(c)）。この手法はデータハザードの回避にも有効である。このような命令入れ替えを命令スケジュールと呼ぶ。RISCのアセンブラは命令スケジュールを当然のように行っている（禁止の設定も可能）。つまり，アセンブラが勝手に最適化するので，プログラマが書いたとおりの順序でコード生成が行われるとは限らないのである。この事実を知ったときは少々衝撃を受けたがいまでは慣れてしまった。

　RISCは制御構造の単純化を目標としているからインタロックは歓迎すべきものではない。ロード遅延をそのまま許し，アセンブラによる命令スケジュールによってのみストールを回避しようという考えがある。これが遅延ロードである。MIPSのR2000/R3000は遅延ロードを許すアーキテクチャを採用している。ただし，R4000からはインタロックするアーキテクチャに変更された。これは，現実問題として，命令の並べ替えができる場合が少なく，多くの場合NOP命令が挿入されてしまうからであろう。NOP命令の挿入により，全体としての命令処理は1クロック余分にかかるが，これはストールで1クロックインタロックしても同じである。それならNOP命令がない分，命令コードのサイズを小さくできる。

(4)制御ハザードと遅延分岐，分岐予測

パイプライン処理を乱すハザードにはデータハザードのほかに制御ハザードがある。これは分岐によるハザードである。ブランチハザードともいう。RISCでは条件分岐は汎用レジスタの値で分岐条件を決定する。MPUによってはCISCと同じく条件フラグを採用しているものもある。この場合の制御ハザードはフラグハザードともいう。さて，条件分岐の場合，分岐条件が確定するまで分岐先の命令フェッチができない（図9(a)）。これによるストールは命令スケジュールで回避することはできず，性能に与える影響が大きい。なお，条件分岐で分岐条件が成立して分岐することをTAKEN，分岐条件が成立せず分岐しないことをNOT TAKEN（あるいはNO TAKEN）という。

パイプライン処理を乱さないため，ストール期間中も（通常は無効化してしまう）分岐命令の後続命令（これを遅延スロットという）を実行させるという考えがある。図5に示すパイプラインではEXステージでTAKEN/NOT TAKENが決定される。したがって分岐先の命令フェッチは1クロックのストール後に実行可能である。TAKENする場合，通常なら分岐命令の後続命令は実行を禁止しなければならない。しかし，その遅延スロットの命令を実行してから，分岐先の命令をフェッチする構造にすればパイプラインはストールしない（図9(b)）。これを遅延分岐と呼ぶ。

このような遅延スロットを設ければ命令スケジュールを行うことができる。分岐命令の前方にある命令を遅延スロットに持ってくることで分岐命令によるストールはなくなる。ただし，遅延スロットに入れる適当な命令がない場合はNOP命令を入れることになる。R2000/R3000のパイプラインはこのようになっているが，現実問題としては，分岐命令の分岐先アドレスもEXステージで計算される（したがって分岐条件を判断するための専用の演算器が別個に必要である）ため，それとほぼ同時に分岐先を命令フェッチするのはタイミング的に厳しい。動作周波数を向上させるためには，遅延分岐を採用しつつも，もう1クロック遅らせるのが望ましい（図9(c)）。まあ，ここら辺をうまく回避するのが回路設計技術ということもできるが。

さて，制御ハザードではTAKENの決定が遅いほどストール期間が長くなる。これはステージ数の多いパイプラインで顕著になる。たとえば，可変長命令を採用するMPUにおいては命令デコードに時間がかかる。パイプラインで少なくとも2ステージ分が必要である。たとえば，

IF RF1 RF2 EX MEM WB

の6ステージからなるパイプラインを考える。TAKENの決定はEXステージなので，これまでの説明より1クロック遅いことになる。このとき分岐命令でのストールは2クロックである（図10）。1クロックを遅延スロットで埋め合わせるとしても，さらに1クロックだけ処理に余計な時

図8 遅延ロード

(a) 遅延ロード

	1	2	3	4	5	6	7
ロード命令	IF	RF	EX	MEM	WB		
結果を参照		IF	RF	EX	MEM	WB	

フォワーディング不可能

(b) インタロック

	1	2	3	4	5	6	7
ロード命令	IF	RF	EX	MEM	WB		
結果を参照		IF	RF	RF	EX	MEM	WB

フォワーディング
ストール

(c) NOP命令を挿入

	1	2	3	4	5	6	7
ロード命令	IF	RF	EX	MEM	WB		
NOP		IF	RF	EX	MEM	WB	
結果を参照			IF	RF	EX	MEM	WB

フォワーディング

図9 遅延分岐

(a) 制御ハザード

	1	2	3	4	5	6	7
分岐命令	IF	RF	EX	MEM	WB		
分岐先命令			IF	RF	EX	MEM	WB

条件確定
ストール

(b) 遅延スロットを設けるとストールがない

	1	2	3	4	5	6	7
分岐命令	IF	RF	EX	MEM	WB		
遅延スロット		IF	RF	EX	MEM	WB	
後続命令(NOT TAKEN)（遅延スロットの次）			IF	RF	EX	MEM	WB
分岐先命令(TAKEN)			IF	RF	EX	MEM	WB

(c) タイミングを緩くするとストールが生じる

	1	2	3	4	5	6	7	8
分岐命令	IF	RF	EX	MEM	WB			
遅延スロット		IF	RF	EX	MEM	WB		
分岐先命令				IF	RF	EX	MEM	WB

ストール

間がかかる。あとで述べるスーパーパイプラインではEXステージより前のステージ数がさらに増加し，分岐命令のストールによる性能低下は深刻なものとなる。

分岐命令の処理を高速化するために分岐予測という機構が採用される。これは，分岐先アドレスをパイプラインのより早いステージで生成し，分岐先の命令フェッチを早期に行う手法である。具体的には，分岐ターゲットバッファ（BTB：Branch Target Buffer），または分岐予測テーブル（BPT：Branch Prediction Table，BHT：Branch History Table）と呼ばれるキャッシュを用意し，分岐命令のアドレス，分岐履歴情報，予測される分岐先アドレスを格納しておく。命令フェッチ時（IFステージ）にBTBを参照し，ヒット（登録してある分岐命令のアドレスと命令フェッチアドレスが一致）すれば，分岐履歴情報に従って，分岐先アドレスを出力し，命令フェッチを行いながら，TAKEN/NOT TAKENの判定を待つ。予測が成功すればフェッチした命令をそのままデコードすればよい。

予測が失敗すれば，実際にEXステージで計算されるアドレスから命令フェッチをやり直し，BTBの分岐履歴情報を更新する（図11）。BTBにヒットし予測が成功する場合はストールがなくなる。BTBにヒットしない場合は，分岐予測を行わない場合と同じタイミングで分岐命令が処理されるが，BTBにヒットするのに予測が失敗する場合は，なにもしない場合に比べて，パイプラインの回復処理にかえって時間がかかってしまうことがある。これが，分岐予測失敗時のペナルティである。したがって，分岐予測を採用しても，予測が失敗ばかりすると，かえって性能が低下するのでヒット率を向上させるための工夫が必要である。図11のパイプラインのモデルではBTBにヒットすると予測した分岐先アドレスから命令フェッチを行うが，MPUによっては（予測する）分岐先の数命令をBTBに格納しておき，そこから命令をフェッチする方法を採用する。こうすることにより，パイプラインは予測していない方向の命令も同時にフェッチできるので，分岐予測が失敗した場合のペナルティを最小化できる。また，分岐予測の成功する確率が高いと思われる場合は，TAKEN/NOT TAKENが決定するまで，予測した分岐先から命令をどんどん先取り（プリフェッチ）する手法もある。パイプラインのステージ数が大きく，TAKEN/NOT TAKENの決定がパイプラインの遅い（後段の）ステージで行われる場合，予測が成功すれば効果的である。逆に予測が失敗したときのペナルティは大きくなる。分岐予測の成功率によほどの自信があるか，失敗時の回復処理がかなり高速化されてないととれない方式であるが，最近のMPUでは結構ポピュラーである。

予測の方法は分岐履歴情報による場合が多い。これは分岐する確率を示す1～2ビットのフラグであり，BTBに登録されている分岐命令ごとに存在する。分岐履歴情報が1ビットの場合は1であるとき「分岐する」，0であるとき「分岐しない」と予測する。これは，その分岐命令が過去1回で分岐したか否かを示している。つまり，以前分岐した分岐命令は今回も分岐すると予測するわけである。分岐履歴情報が2ビットの場合はもう少し慎重である。ビット列の意味の持たせ方はいろいろ考えられるが，たとえば，11，10で「分岐する」，01，00で「分岐しない」と予測する。これは，その分岐命令が，過去2回において何回「連続して」分岐したかを示す。分岐する傾向が大きい方向に予測するわけだ。なお，分岐する（と予測する）分岐命令のみをBTBに登録する方法もある。この場合は分岐履歴情報は不要で，BTBにヒットすれば「分岐する」，ヒットしなければ「分岐しない」と予測する。この場合，分岐予測が成功する確率は，分岐履歴情報が1ビットの場合とほぼ同等であるが，BTBの回路規模は約半分になる。

分岐予測を行わない場合で，分岐命令を高速化する方法として，分岐先と分岐元の命令を同時にプリフェッチする手法もある。というか，それに関係する特許は，昔，山のようにあった。これは回路規模が大きくなるため，あまり現実的でない。といいつつも，インテル系のMPU（特にIA-64）ではそのような説明をよく目にする。ただし，具体的な実装方法は不明である。やはり，特許が絡んでいるのか。

(5)構造ハザード

構造ハザードとはパイプラインの2つ以上のステージが1個しかないハードウェア資源を取り合うために生じるハザードである。たとえば，5ステージで構成されるパイプラインでは1時刻に5つすべてのステージが実行される可能性がある。もし，各ステージで同一の演算器などを使用する場合は競合するので優先されるステージ以外は待ち合わせをする必要がある。RISCの場合，ほとんどのハードウェア資源は競合しないように設計されているのであまり問題はない。しかし，例外もある。それはキャッシュ（あるいはメモリ）である。図5(b)を見ていただきたい。時刻4において命令1のMEMステージと命令4のIFステージが重なっている。もし，命令1がロード/ストア命令であり，命令とデータキャッシュの区別がなく単一のキャッシュしかない場合は，IFステージもMEMステージもキャッシュアクセスなので，資源の競合が生じる。キャッシュが存在しない場合もメモリの競合が生じる。この場合は，先にある命令1のMEMステージを優先させ，命令4のIFステージをインタロックして待ち合わせることになる。これは，できるだけパイプラインをインタロックさせないというRISCの考え方に反する。

そこで，多くのMPUでは命令とデータを2つのキャッシュに分割して同時にアクセスできるようにしている。これならアクセスの競合によるインタロックは発生しない。このように命令とデータの供給経路を独立に取る方式をハーバードアーキテクチャという。なお，命令とデータに関しては，TLBがひとつしかない場合，アドレス変換時にも資源の競合が生じる。それを避けるため命令用とデータ用のTLBを独立に用意するアーキテクチャもある。多くの場合，命令はアクセスするアドレス範囲が小さい（あるいは連続している）ため，命令用のTLBをマイクロTLBとして，仮想アドレスと物理アドレスのペアを本当のTLBからキャッシュして持っているのが普通である。

(6)CPIとMIPS値

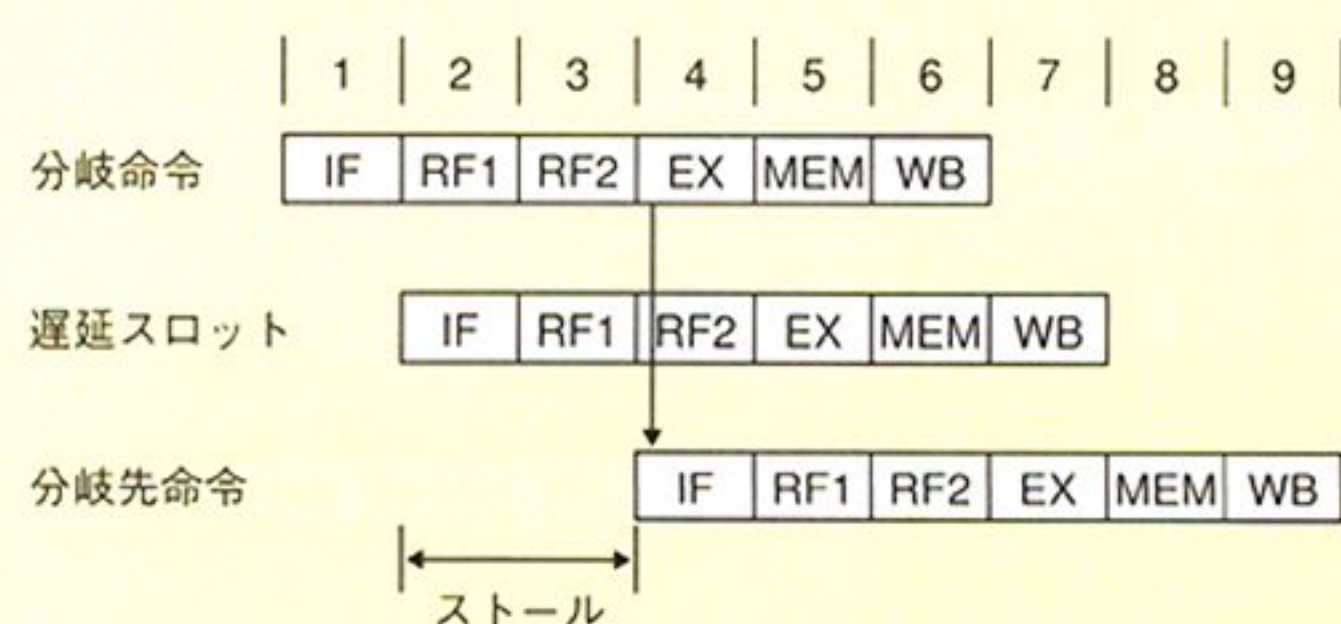
パイプライン処理における命令の実行効率を表す指標としてCPI（Clock cycles Per Instruction）がある。これは1命令を実行するのに必要なクロック数である。RISCの当初の目標はキャッシュと効率的なパイプライン処理でCPIを1にすることにある。実際，RISCはキャッシュにヒットしパイプラインにインタロックがない場合はCPIが限りなく1に近づく。インテルのMPUの平均CPIに関しては，i486を設計した技術者のひとりであるPatrick Gelsingerのレポートがある。それによると次のような値が出ている。

図10　命令デコードが2ステージの場合の制御ハザード

	1	2	3	4	5	6	7	8	9
分岐命令	IF	RF1	RF2	EX	MEM	WB			
遅延スロット		IF	RF1	RF2	EX	MEM	WB		
分岐先命令				IF	RF1	RF2	EX	MEM	WB

（時刻1～3：ストール）

図11　分岐予測

		1	2	3	4	5	6	7	8	9	
分岐命令		IF	RF1	RF2	EX	MEM	WB				
遅延スロット			IF	RF1	RF2	EX	MEM	WB			
			TA 遅延先アドレス（予測）								
分岐先				IF	RF1	RF2	EX	MEM	WB		予測成功
					IF	RF1	RF2	EX	MEM	WB	予測失敗

8086	15.0
80286	6.0
80386	4.5
i486	1.7

MPUが進化するにつれてパイプラインの効率が上昇しているのがわかる。さすがインテルというところか。i486でCPIが急激に改善したのはキャッシュの恩恵といわれている。CISCでありながらRISC並みのパイプライン処理を採用したことも一因であろう。現在のPentiumのCPIは0.6～0.7であるという(ちょっと性能がよすぎる感もするが)。これは次回で説明するスーパースカラの恩恵である。

CPIはMIPS(Million Instructions Per Second)値と密接な関係がある。MIPS値とは1秒間に実行できる命令数(100万命令単位)であるから，動作周波数とCPIが決まれば，

周波数(MHz単位)÷CPI

という計算式でMIPS値が求まる。この式で，上のx86プロセッサのMIPS値を計算すると次のようになる。

8086	5MHz	→	0.33MIPS
80286	8MHz	→	1.33MIPS
80386	12MHz	→	2.67MIPS
i486	25MHz	→	14.71MIPS
Pentium	66MHz	→	110MIPS

まあ，実際に公表されるMIPS値はDhrystone MIPS(最近ではDMIPSと略記されることもある)なのでもう少し高い値になっているかもしれない。これは，Dhrystoneベンチマークを実行した性能が，1MIPS相当のVAX-11/780の何倍であるかを表すものである。Dhrystone MIPSでは，シングルパイプラインのMPUのCPIが1を割ることも多く，直感的ではない。しかし，現在実際に使用されているMIPS値はDhrystone MIPSなので慣れが必要である。もっとも，x86系のMPUはMIPS値の公表をやめてしまっている(表向きの理由はいろいろあるが，発表するとCPIの大きさが問題となるからだろう)ので，性能を比較するには動作周波数に頼るしかない。各メーカーは独自の基準で従来品との相対性能を公表しているが，異なるメーカー間での性能比較はできない。いくら動作周波数が高くてもCPIが悪ければなんにもならないのだが，メーカーやマスコミはこの点を意図的にうやむやにしているように思える。

●スーパーパイプライン

MPUを高い周波数で動作させるためには，パイプラインの1ステージ当たりで実行する論理を減少させる必要がある。単純に考えると従来1ステージで実行していた処理を2ステージに分割することである。つまり，高速な動作周波数になるにつれてパイプラインのステージ数が増加する傾向にある。いま，パイプラインのステージを，

IF1 IF2 RF1 RF2 EX1 EX2 MEM1 MEM2 WB1 WB2

としてCPIを試算してみよう。図12(a)では4命令を13クロックで実行しているのでCPIは3.25である。一方，図12(b)では4命令を8クロックで実行しているのでCPIは2.0である。スーパーパイプライン構成にすることでCPIは約1.5倍に増加する。しかし，動作周波数を2倍に引き上げることができれば実質的な性能は向上する。これがスーパーパイプラインの考え方である。

スーパーパイプラインを最初に採用したのはMIPSのR4000である。これは当初100MHz動作であったが，最近では250MHz動作を達成している。ほぼ同時期に登場したDECのAlpha(21064)は200MHz動作を達成していた。これは1990年代の始めとしては驚異的な動作周波数だった。このため，Alphaは世界最高速のMPUとしてギネスブックに登録された。

最近では，

動作周波数を上げる＝パイプラインのステージ数を増やす

という図式が常識のように語られるようになった。インテルのPentium4(Willamette)は20ステージのパイプライン構成で1.5GHz以上の動作を目指している。IPコアの分野でも，Lexra社がLX4189でパイプラインを従来の5ステージから6ステージに変更することで，初めて250MHz以上の動作周波数を達成したと発表した。まあ，動作周波数を高速にするためにはパイプラインのステージ数を増加する必要があるのは本当だが，逆は必ずしも真ではないと思うので，そんな単純なものではないと思うのだが。しかし，これからのMPU設計においては，パイプラインのステージ数を増加して動作周波数を稼ぎ，それによるIPCの低下は分岐予測を高度にすることで補っていく傾向になるのは間違いないだろう。

●プリフェッチとデカップル(decouble)構成

CISCにおいてプリフェッチが有効であることを述べたが，RISCにおいてもプリフェッチが有効な場合がある。命令フェッチが命令キャッシュにヒットする限りは，各サイクルごとに命令デコーダに命令が供給されるので，プリフェッチをして命令をFIFOなどに蓄えておく必要はない。しかし，命令キャッシュミスが発生すると命令供給が停止するので，パイプラインがストールしてしまう。それを防ぐためにプリフェッチは有効である。命令デコード以降のパイプライン処理とは独立に命令を絶えずプリフェッチしておけば命令デコードにおいて命令の供給が停止する頻度は少なくなる。

命令キャッシュのミスが発生した場合，命令キャッシュへの書き込みと同時にデコーダへ命令をバイパスする「命令ストリーミング」もパイプラインのストールを低減させる方法のひとつである。しかし，命令ストリーミングでは，(通常は)パイプラインクロックよりも遅いバスクロックに同期して命令供給が行われるので，命令ストリーミング中の命令処理はバスクロック同期に近くなり，効率があまりよくない。プリフェッチは，命令キャッシュミスの発生が契機となるわけではなく，無条件に命令フェッチを行っていくので，命令ストリーミングよりも効率がいい(はずである)。

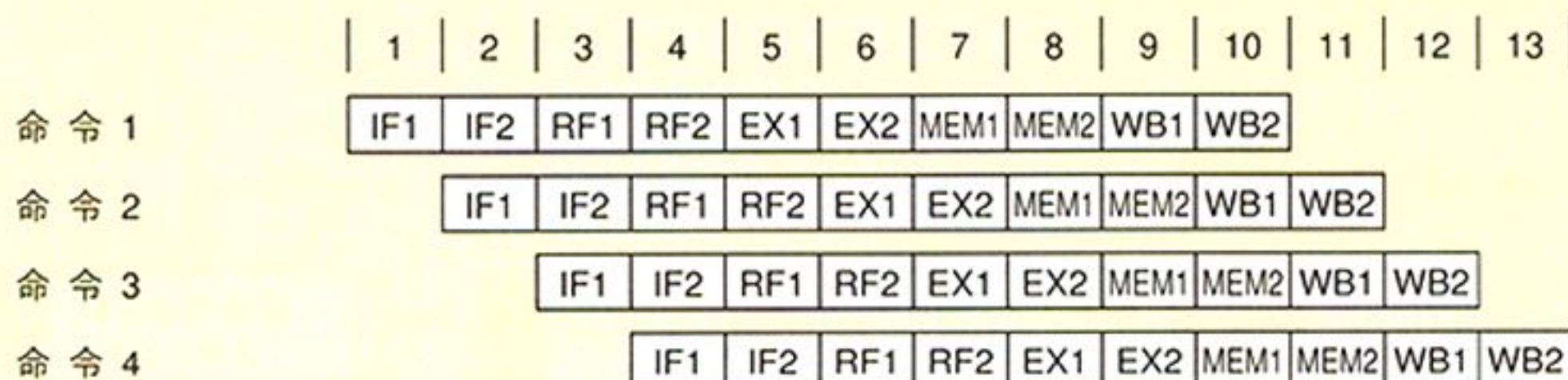
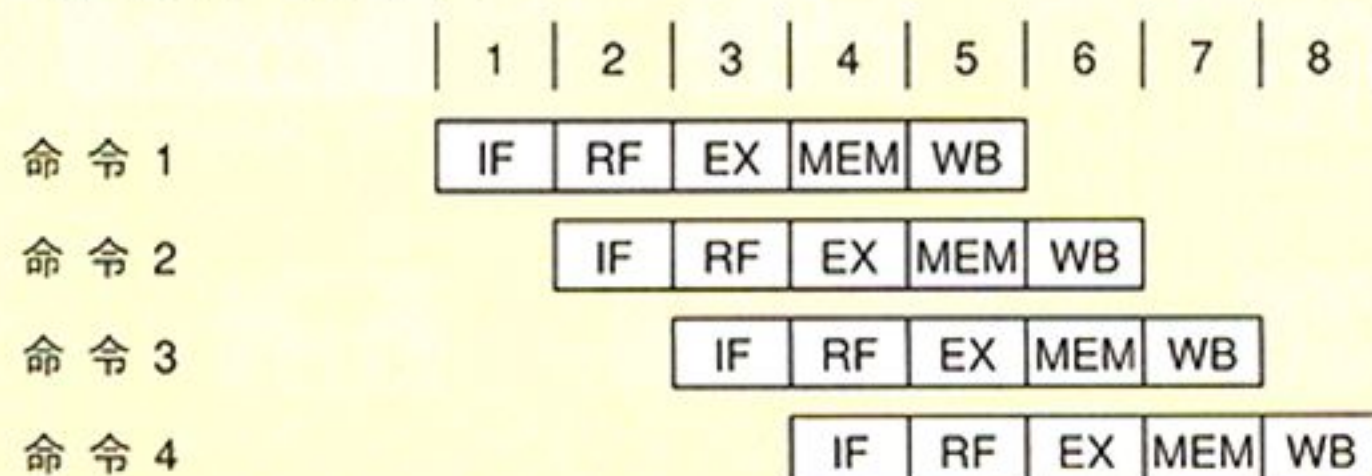
シングルパイプラインではあまりお目にかからないが，デカップル方式という構成がある。これは，プリフェッチとよく似た概念であるが，命令デコードと実行ステージの中間にFIFO(First In First Outのバッファ構造)を置いて，そのFIFOに絶えずデコード済みの命令を格納しておく。こ

図12 スーパーパイプラインの効率

(a) スーパーパイプライン

	1	2	3	4	5	6	7	8	9	10	11	12	13
命令1	IF1	IF2	RF1	RF2	EX1	EX2	MEM1	MEM2	WB1	WB2			
命令2		IF1	IF2	RF1	RF2	EX1	EX2	MEM1	MEM2	WB1	WB2		
命令3			IF1	IF2	RF1	RF2	EX1	EX2	MEM1	MEM2	WB1	WB2	
命令4				IF1	IF2	RF1	RF2	EX1	EX2	MEM1	MEM2	WB1	WB2

(b) 通常のパイプライン

	1	2	3	4	5	6	7	8
命令1	IF	RF	EX	MEM	WB			
命令2		IF	RF	EX	MEM	WB		
命令3			IF	RF	EX	MEM	WB	
命令4				IF	RF	EX	MEM	WB

うすることで，FIFO内の命令はソースオペランドが有効である限り，各サイクルごとに命令実行を開始することが可能になる。つまり，オペランドの依存性による命令デコードステージでのストールが見えなくなる(パイプライン効率が上がる)。当然，命令フェッチとデコードまでのステージと実行ステージ以降は別のクロックに同期して独立に動く。パイプラインがデコードまでと実行以降とに分離(decouple)されていることでデカップル方式と呼ばれる。

デカップル方式の利点は，命令デコードを行うので分岐命令を認識することが可能であり，分岐予測をしながら投機的(speculative)に命令のプリフェッチを行うことができる点である。単なるプリフェッチであれば，分岐命令以降にある命令を無駄にプリフェッチする恐れがある。分岐予測に従ってプリフェッチを行うことができれば，(分岐予測が当たる限り)命令フェッチのロスはなくなる。このため，デカップル方式では，分岐予測が有効に働けば，パイプライン処理の中で，命令フェッチと命令デコードステージを無視することができる。たとえば，5ステージのパイプライン処理ならば，2ステージ少ない，3ステージのパイプラインと同等の効率で命令を処理できる。

プリフェッチやデカップル方式での投機的なデコードは，実行ステージ以降で発生するパイプラインストールの合間を縫って行われる。実行ステージ以降でストールがまったく発生しなければ，プリフェッチ機構自身が無意味なものになってしまう。パイプライン効率は落ちないが，プリフェッチをしてもしなくても同じ効率にしかならないので，余分な回路ということになる。実際問題として，シングルパイプラインではロード遅延とデータキャッシュミス以外では実行ステージ以降でのストールは発生せず，プリフェッチの回路規模の割には性能は向上しないと思われる。しかし，2命令以上を同時に処理するスーパースカラにおいては，命令デコードの倍以上の速度で命令が処理されていくので，プリフェッチや投機的デコードの機構を用意しておかなければ命令供給が命令消費に追いつかなくなる。

●命令書き換えとパイプライン

昔，8086や68000というMPUが全盛だった頃，プログラムのコードサイズを削減するために，命令コード領域をストア命令で書き換えて実行するという技が重宝されていた。これは自己書き換えと呼ばれる手法である。自己書き換えはパイプラインを採用するMPUでは期待どおりの動作をするとは限らない。それは，パイプラインのステージを考えれば明らかで，書き換えた命令のフェッチ(IF)は書き換える命令のライト(WB)以降でなければならないためである。たとえば，

```
IF RF EX MEM WB ↘
                 IF RF EX MEM WB
```

という5ステージ構成では，最低5命令以後を書き換えなければ，そこを正しくフェッチできない。また，命令のプリフェッチを行う場合は，一概に何命令後を書き換えると大丈夫かということは保証できない。書き換えた場所にジャンプすればよいという考えもある。この方法も，分岐予測などで命令フェッチが先行する場合はうまくいかないことがある。

ところで，最近のMPUは命令キャッシュとデータキャッシュが分離されているので，単純には命令コードを書き換えることはできない。ストア命令を実行してもデータキャッシュの内容が変更されるだけで，命令キャッシュの内容は変わらないからである。ただし，(OSに限られるが)特権命令を使えば，書き換えたアドレスに対応する命令キャッシュの内容を無効化することで，自己書き換えを実現できる。もし，ライトバックキャッシュ構成ならデータキャッシュを最初に強制的にライトバックさせることも必要である。

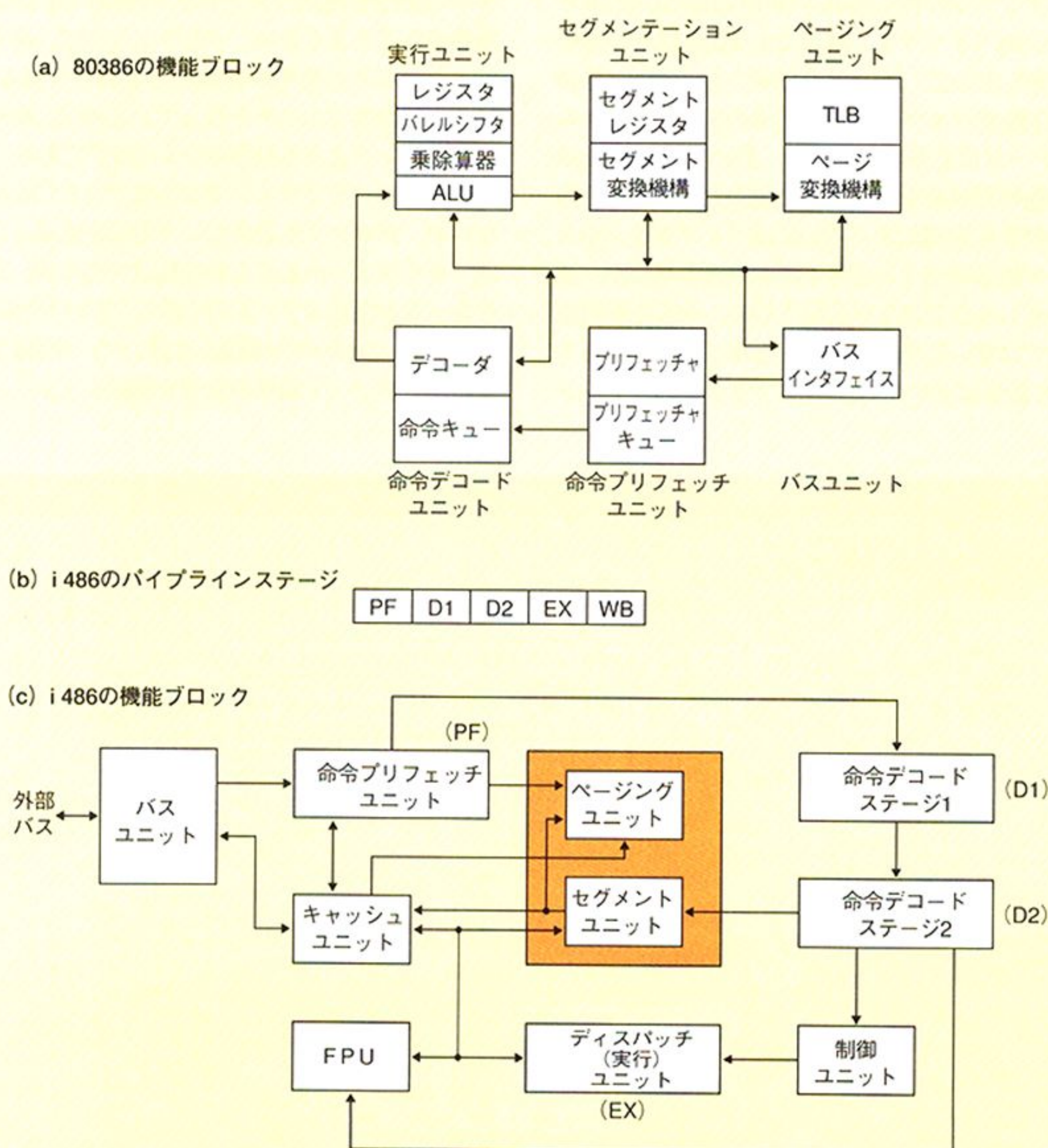
と，自己書き換えを推奨するような説明をしてみたが，最近のプログラミングでは好ましくないものとされている。MMUが内蔵され，十分大きなアドレス空間を使ってプログラムを作ることが可能なので，わざわざプログラムの流れをわかりにくくする自己書き換えを行う理由はない。とはいえ，仮想記憶のデマンドページングで行われるスワップインは壮大な命令書き換えではないかと考えると，OSなら自己書き換えをしてもいいのかと突っ込みたくなる。

図13　80386/i486のパイプライン

●パイプライン処理の実際

それでは実際のMPUのパイプライン処理を見ていこう。あまり資料がないので正確さを欠くかもしれないが，気分としてわかっていただけば幸いである。まず，超有名なx86の代表として80386とi486，教科書的なパイプラインの代表としてR3000/R4000を取り上げる。そのあとに，組み込み制御用途に新たに生み出された低価格，低消費電力RISCの代表格として，SHシリーズ，ARM，V800シリーズのパイプラインにも言及する。

(1)80386/i486

80386のパイプラインについての文献は少ない。ある資料によると，80386のパイプラインは次の6つのユニットを同時に実行する6ステージパイプラインということである。

1. バスユニット：命令プリフェッチユニット，実行ユニットからのプリフェッチ要求を優先度をつけて受け付ける。必要に応じてバスサイクルを発生させ，メモリ，I/Oにアクセスする
2. 命令プリフェッチユニット：バスユニットがアイドルのとき，命令のプリフェッチを行い16バイトのプリフェッチキューに入れる
3. 命令デコードユニット：プリフェッチキューから命令を取り出し，3エントリのデコード済みキューに入れ，実行ユニットでの実行を待つ
4. 実行ユニット：デコード済みキューの命令を順次取り出して実行する。実行と同時に次の命令のためのメモリ参照を並行して行う。
   このユニットは次の3つのサブユニットで構成される
   (i) 制御サブユニット：マイクロコードROM，乗除算器，実行アドレス生成器で構成される
   (ii) データサブユニット：ALU，レジスタファイル，バレルシフタで構成される
   (iii) 保護サブユニット：セグメンテーション違反の検出を行う
5. セグメンテーションユニット：実行ユニットの指示に従って，2つのアドレス情報を32ビットのアドレスに変換する。キャッシュ機能（TLBみたいなもの？）があるらしい
6. ページングユニット：仮想アドレスを物理アドレスに変換する。TLBを備え，高速なアドレス変換を実現する。物理アドレスはバスユニットにも送られメモリやI/Oアクセスが行われる

各ユニットの関係を図13(a)に示す。6ユニットが同時に実行されるといっても，機能的な関わりを見る限り，これらが逐次的に実行されるとは思えない。実際には，

プリフェッチ→命令デコード→実行（→ライトバック）

の3（または4）ステージ構成ではないかと推測される。基本的には次に示すi486と同じであると思われる。

i486のパイプラインはPF，D1，D2，EX，WBの5ステージで構成される。これを図13(b)に示す。また，パイプラインに関係する機能ブロックの関係を図13(c)に示す。各ステージでの処理は次のようになっている。

PF　命令プリフェッチ。バスユニット，またはキャッシュユニットから命令をプリフェッチする。バスやキャッシュがアイドルな命令をプリフェッチし続ける。

D1　命令デコードの第1段目。可変長の命令コードの位置合わせを行いデコードする。

D2　命令デコードの第2段目。命令デコードを完了する。同時にオペランドのアドレス計算とオペランドリードを行う。

EX　キャッシュアクセス，または命令実行をする。

WB　命令実行結果をキャッシュまたはレジスタに書き込む。

i486のパイプラインに関してはユーザーズマニュアルにも詳細な記述はないのであまり詳しいことはわからない。もっとも，デコードがD1，D2の2ステージに分かれているのは可変長の命令を効率よくデコードするためという記述をどこかで読んだ気もするが。i486はキャッシュをチップ内に内蔵し，ほとんどの命令をハードワイヤードロジックにより1クロックで実行するなどRISCの要素が盛り込まれている。

なお，i486のユーザーズマニュアルによると，i486のパイプラインは命令プリフェッチ，命令デコード，マイクロコード実行，整数演算，浮動小数点演算，セグメンテーション，ページング，キャッシュ管理，バスオペレーションが同時に実行される，とある。上で述べたパイプラインステージとの関連がはっきりしない点が（やはり）CISCのパイプラインといえるかもしれない。また，多くの1クロック実行のRISCに対する利点として，命令実行中に次の命令で使用するデータをキャッシュからリードできるので，（データ転送の行われる）EXステージより前のD1ステージからデータを利用できる点が挙げられている（この仕組みは80386と同じである）。この方式の欠点は，直前の命令が変更するレジスタの値をアドレスとしてメモリアクセスをする場合には3クロックかかる点である。しかし，多くのメモリアクセスは値が固定のスタックポインタやフレームポインタからの相対アドレスで行われるため，性能に与える影響は少ないとされている。

(2)R3000

R3000のパイプラインは基本的には図5で示したRISCのパイプラインと同じである。IF，RF，EX，MEM，WBの5ステージで構成される。実際には$\phi 1$，$\phi 2$の2相クロックで動作し，1クロック間に2ステップの処理を行っている。図14にR3000のパイプラインの詳細を示す。各ステージでの動作は次のようになっている。

IF $\phi 1$　マイクロTLB（ITLB）を使用して命令の仮想アドレス（IVA）を物理アドレスに変換する。分岐先アドレスはRFステージの$\phi 2$で計算され，EXステージの$\phi 1$でアドレス変換される。

IF $\phi 2$　物理アドレスを命令キャッシュに転送し，命令キャッシュをアクセスする（ICache）。

RF $\phi 1$　命令キャッシュのヒット/ミスがチェックされ，命令キャッシュから命令を読み出す（ICache）。

RF $\phi 2$　命令をデコードする（ID）。
分岐命令の場合は分岐先アドレスを計算する。
レジスタファイルをリードする（RF）。

EX $\phi 1+\phi 2$　オペランドをほかのパイプラインステージからバイパスし演算する（ALU）。ストアするデータがあれば位置合わせを行う。

EX $\phi 1$　分岐命令ならTAKEN/NOT TAKENを決定する。
ロード/ストア命令ならオペランドの仮想アドレスを計算する（DVA）。

EX $\phi 2$　ロード/ストア命令ならTLBを使用してオペランドの仮想アドレスを物理アドレスに変換する（DTLB）。

MEM $\phi 1$　ロード/ストア命令なら物理アドレスをデータキャッシュに転送し，データキャッシュをアクセスする（DCache）。

MEM $\phi 2$　データキャッシュのヒット/ミスがチェックされ，命令キャッシュからオペランドを読み出す（DCache）。

WB $\phi 1$　EXステージでの演算結果をレジスタファイルに書き込む（WB）。ストア命令の場合はデータキャッシュに書き込む。

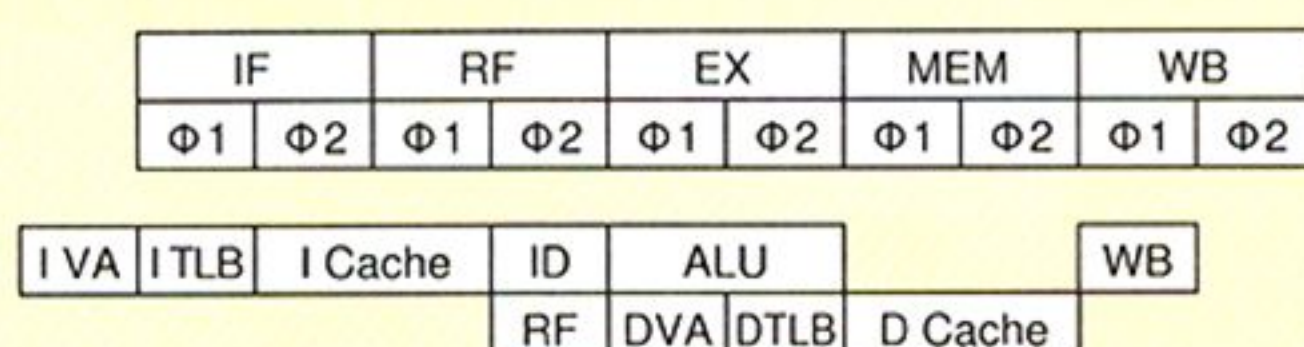
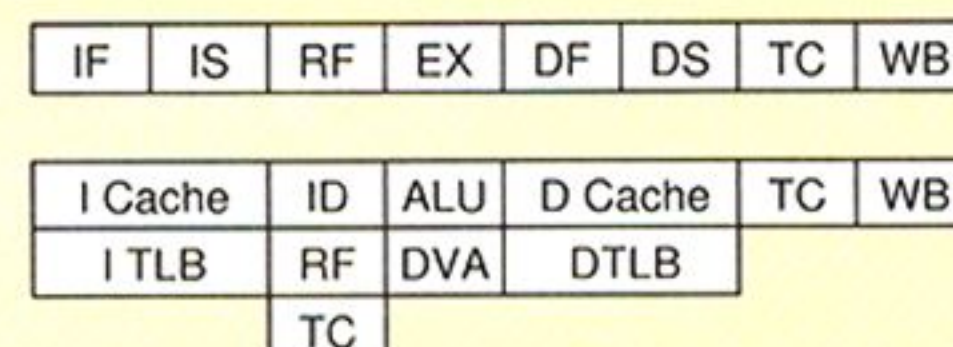
図14　R3000のパイプライン

IF		RF		EX		MEM		WB	
Φ1	Φ2	Φ1	Φ2	Φ1	Φ2	Φ1	Φ2	Φ1	Φ2

I VA	I TLB	I Cache	ID	ALU		WB
			RF	DVA	DTLB	D Cache

図15　R4000のパイプライン

IF	IS	RF	EX	DF	DS	TC	WB

I Cache	ID	ALU	D Cache	TC	WB
I TLB	RF	DVA	DTLB		
	TC				

R3000のアーキテクチャであるMIPSとはMicroprocessor without Interlocked Pipeline Stagesの省略形である。パイプラインステージをインタロックしないマイクロプロセッサという意味である。その名称のとおり，R3000はキャッシュミスが発生した場合のメモリからデータをリードしてくる場合のストール期間以外はパイプラインがインタロックしない。そのために，遅延ロード，遅延分岐を積極的に採用している。このため，キャッシュにヒットし続ける限りはCPIの値がほぼ1を保ち続ける。ただし，例外もある。乗除算命令は1クロックで処理をすることができない。これらの命令があるとインタロックが発生する可能性がある。しかし，なるべくインタロックさせないための工夫が凝らされている。乗除算の結果は汎用レジスタではなく，専用のレジスタに格納される。また，乗除算器はほかの命令のパイプライン実行と並行して動作する。つまり，乗除算命令では汎用レジスタ間のデータハザードは発生しない。このため乗除算命令の処理は通常のパイプライン動作に影響を与えない。乗除算が完了したあとで，専用レジスタから演算結果を取り出せば(専用レジスタから汎用レジスタへの転送命令が用意されている)インタロックは発生しない。R3000では乗算と除算の実行時間が，それぞれ12クロック，35クロックである。乗算命令に関していえば実行を開始してから12クロック後に結果を取り出せばインタロックは発生しない。プログラム的には乗算命令と結果を取り出す命令の間が12命令分空いていればよい。一方，12クロック未満で結果を取り出そうとすると，アーキテクチャ的には不本意ながらインタロックしてしまう。まあ，現実的には乗除算命令と結果を取り出す命令の間はせいぜい3命令程度しか空けることができないので，乗除算命令があるとほとんどの場合インタロックしてしまうのだが。

(3)R4000

R4000はスーパーパイプライン構造を採用し，高い動作周波数で動作させることを目的としている。パイプラインはIF，IS，RF，EX，DF，DS，TC，WBの8ステージで構成され，(多分)単相クロックに同期して動作する。図15にR4000のパイプラインの詳細を示す。各ステージでの動作は次のようになっている。

- IF　命令フェッチ1段目。命令の仮想アドレスが命令キャッシュとTLBに転送される
- IS　命令フェッチ2段目。命令キャッシュが命令を出力し，同時にTLBは命令の物理アドレスを出力する
- RF　レジスタファイル。次の3動作が並行して行われる
  1）命令をデコードし，インタロック条件をチェックする
  2）命令キャッシュのヒット/ミスがチェックされる
  3）レジスタファイルからオペランドをフェッチする
- EX　命令実行。次の3動作のひとつが実行される
  1）命令がレジスタ - レジスタ間命令なら演算を実行する
  2）命令がロード/ストア命令ならオペランドの仮想アドレスを計算する
  3）命令が分岐命令なら，分岐先の仮想アドレスを計算する。同時に分岐のTAKEN/NOT TAKENを決定する
- DF　データキャッシュ1段目。オペランドの仮想アドレスがデータキャッシュとTLBに転送される
- DS　データキャッシュ2段目。データキャッシュが値を出力する。同時にTLBはオペランドの物理アドレスを出力する
- TC　タグチェック。ロード/ストア命令の場合，データキャッシュのヒット/ミスをチェックする
- WB　ライトバック。命令の実行結果をレジスタファイルに書き込む。ストア命令の場合はデータキャッシュに書き込む

図16　Jade(4Kc)とOPal(5Kc)のパイプライン

(a) R3000のパイプラインとJadeのパイプライン (Micro Processor Reportより)

I Fetch	Decode	Execution	Memory	Write bach

R3000: IA2 / I Cache Rd / IA1 / ITLB / RF Rd / ALU Op / DA / DTLB / D Cache / LA / RF Wr

Jade: I Cache Rd / IA1 / IA2 / ITLB / RF Rd / ALU Op / DA / DTLB / D Cache / LA / RF Wr

(b) OPalのパイプライン

IF		Dispatch		RF		EX		MEM		WB	
Φ1	Φ2	Φ1	Φ2	Φ1	Φ2	Φ1	Φ2	Φ1	Φ2	Φ1	Φ2

I Cache / ITLB / WS / Dispatch / MIPSI6 / IA1 / IA2 / RF / ID / ALU / DA / DTLB / D Cache / WS / LA / WB

R4000ではパイプラインが8ステージになったため，分岐命令の実行時に3クロック，ロード命令の実行時に2クロックの遅延スロットが生じる。分岐命令においてはR3000と互換性を持たせるため遅延スロットの1命令分は実行するが，残りの2クロックはバブル(無駄な時間)になる。分岐命令の実行時間がR3000の1クロックから3クロックになった(遅延スロットを含まない)と思えばいい。ロード命令においては遅延スロットに相当する後続2命令がロード命令のデスティネーションオペランドと一致している場合はインタロックが生じる。つまり，R4000では遅延ロードを採用しない。さすがに，ロード命令とその結果を使用する命令の間を2命令分も空けるのは現実的でないと考えたのであろう。

分岐命令の実行時間を短縮するため，R4000ではLikely分岐が導入された。Likely分岐とは，分岐条件が成立するときのみ遅延スロットの命令を実行する条件分岐命令である。分岐条件が成立しなければ遅延スロットは無効化される。遅延スロットにNOP命令があると考えてもよい。分岐命令がループ処理の終わりにあるような場合，分岐命令をLikelyにして分岐先の1命令を遅延スロットに置けば，ループ内の命令が1命令減少するので，実質的に分岐命令の実行時間を短縮できる。これは一種の(静的な)分岐予測とみなすこともできる。

(4)Jade(4Kc)/Opal(5Kc)

MIPSはいままでの命令セットアーキテクチャ

を整理し，MIPS32とMIPS64の2種類に統合した。また，同時にそれぞれの命令セットを採用する論理合成可能なIPコアを発表した。それが，Jade (4Kc)とOpal (5Kc)である。MIPSアーキテクチャのパイプラインはR3000で一応の完成を見ている。しかし，R3000のパイプラインは，最大限の性能を引き出すことができるが，非常に厳しいタイミングで設計されており，動作周波数を向上させるのが難しい。その意味でR4000はタイミングに余裕のあるスーパーパイプラインを採用したのは正解であるが，それと引き換えにCPIを増加させてしまった。動作周波数の向上を考慮した全体的な性能向上はほんのわずかである。そのため，R4000の後継ではR3000ライクな5ステージパイプラインに戻されてしまった。JadeとOpalもR3000のパイプラインを基本とし，論理合成が容易なようにパイプラインを変更してある。もっとも顕著な変更はクロックの単相化であろう。パイプラインの説明では2相クロックを思わせる記述もあるが，実際にはクロックの立ち上がりと立ち下がりエッジに同期していると推測される。

図16(a)にR3000とJadeのパイプラインの比較を示す。R3000のパイプラインが図14と一部異なっているが，図14は説明用に簡略化したもの，図16(a)は現実に近いものと理解すればいいだろう。図16(a)に示すように，R3000は多くのクリティカルな操作(命令キャッシュアクセス，レジスタリード，データTLB参照)をパイプラインクロックの立ち下がりエッジに同期して行っている。また，データキャッシュアクセスはクロックの立ち上がり同期であり，データキャッシュからリードしたデータの位置合わせ(図のLA)を同じパイプラインステージ内で行うので，タイミングはかなり厳しい。命令のアドレス計算も，命令キャッシュアクセスの前後に，2つの1/2サイクルのアクセス(IA1，IA2)に分割して行われるので，制御が複雑になる。これらが，IPコアとして容易に論理合成を行うためのボトルネックになっている。

また，SRAM(キャッシュ)のアクセスタイミングも厳しく，キャッシュをメモリコンパイラなどで自動生成するのが困難である。このため，Jadeではパイプラインが再設計された。具体的には，すべての操作を1フェーズ早めてクロックの立ち上がり同期にした。さらに，命令TLBアクセスとデータキャッシュアクセスを1ステージ早くして，リードデータの位置合わせをキャッシュアクセスと別のステージに持っていった。結果として，クリティカルな操作は立ち上がりエッジ同期になった。命令のアドレス計算は，命令キャッシュアクセス後の，ひとつのパイプラインステージに統合された。これらの結果，データキャッシュアクセスのタイミングに余裕ができた。図を見てわかるように，レジスタファイルへのライトを位置合わせの直後(立ち下がり同期)にすることでパイプラインステージ数を5ステージから4ステージにすることも可能である。

しかし，Jadeではクロックの立ち上がり同期にこだわり，結果として5ステージのパイプラインとなっている。Jadeパイプラインは3つの利点があるといわれている。ひとつ目はキャッシュアクセスに余裕がある。2つ目は，クリティカルな操作がすべて立ち上がり同期になっているので，ある論理ブロックをユーザーが設計した論理に置き換えることが容易な点である。3つ目は，論理合成ツールによる遅延の調整が容易になる点である。本来，論理合成を想定した機能設計はクロック遅延のばらつき(クロックスキュー)を一定値内に収める操作を容易にするためにクロックの立ち上がりエッジのみを使用する。これを実践したわけだ。MIPSの発表によると，0.25 μmプロセスで製造した場合の動作周波数は最悪の場合(単純な論理合成)で100～150MHz，典型的な場合(専用設計)で150～255MHzだそうである。クリティカルな操作を立ち上がり同期にしたとはいえ，パイプライン効率はR3000のそれと大差がないのも事実で，この動作周波数が可能なのか否かは実際に回路設計した人にしかわからないだろう。

さて，Opalではさらにパイプラインが変更された。OpalのパイプラインはIF，Dispatch，RF，EX，MEM，WBの6ステージで構成され，φ1，φ2の2相クロックで動作するとされている。しかし，論理合成を容易にするために単相クロックを採用しつつも説明上の方便でφ1，φ2を使用しているのではないかと思われる。図16(b)にOpalのパイプラインを示す。Opal自身はスカラプロセッサだが，スーパースカラへの移行の可能性を残している。つまり，ディスパッチステージが命令フェッチとレジスタリード/命令デコードステージの間に挿入された。このためパイプラインは，Jadeより1ステージ多い，6ステージとなる。これは，将来的には，複数の演算ユニットに命令をディスパッチ(発行)するために使用する。命令デコード自体にも余裕ができるので，動作周波数が少し向上する。また，この追加ステージはMIPS16のためのプリデコードステージとしても利用できる。パイプラインのステージ数が増加することで分岐の性能が悪くなるが，Opalでは静的な分岐予測と命令プリフェッチで対応している。分岐はすべてTAKENするものと仮定し，投機的に6命令をフェッチできる。分岐予測が外れた場合のペナルティは1サイクルにすぎないという(本当かいな)。Opalのパイプラインの詳細を以下に示す。

IF φ1＋φ2	命令キャッシュにアクセスする(ICache)。命令の仮想アドレスはDispatchステージ(IA1)とRFステージ(IA2)で計算される
IF φ2	マイクロTLBにアクセスし命令の仮想アドレスを物理アドレスに変換する(ITLB)
Dispatch	命令キャッシュのヒット/ミスをチェックする(WS：Way Select)
φ1＋φ2	スーパースカラ構成をとるための命令ディスパッチ用のタイミングを提供する(Dispatch) MIPS16をサポートする場合のプリデコードタイミングを提供する(MIPS16) 次の命令のための命令の仮想アドレスを用意する(IA1)
RF φ1＋φ2	レジスタをフェッチする(RF)。命令をデコードする(ID)
RF φ1	分岐先の仮想アドレスを計算する(IA2)
EX φ1＋φ2	演算を行う(ALU)
EX φ1	ロード/ストア命令のオペランドアドレスを計算する(DA)
EX φ2	データキャッシュへのアクセス(DCache)。1段目
MEM φ1	オペランドの仮想アドレスを物理アドレスに変換する(DTLB)。データキャッシュへのアクセス(DCache)。2段目
MEM φ2	データキャッシュのヒット/ミスをチェックする(WS) データキャッシュからフェッチしたデータ，データキャッシュにストアするデータの位置合わせをする(LA)
WB φ1＋φ2	EXステージでの演算結果をレジスタファイルに書き込む(WB)。ストア命令の場合はデータキャッシュに書き込む

MIPSの発表によると，Opalを0.15 μmプロセスで製造した場合の動作周波数は450MHz，0.18 μmプロセスでは375MHzだそうである。Opalでは，Jadeでわざわざ立ち上がり同期に揃

図17 SH1/SH2/SH3のパイプライン

レジスターレジスタ間演算(SH1/2)	IF	ID	EX		
(SH3)	IF	ID	EX	ma	WB
ロード命令	IF	ID	EX	MA	WB
ストア命令	IF	ID	EX	MA	

図18　ARM構成

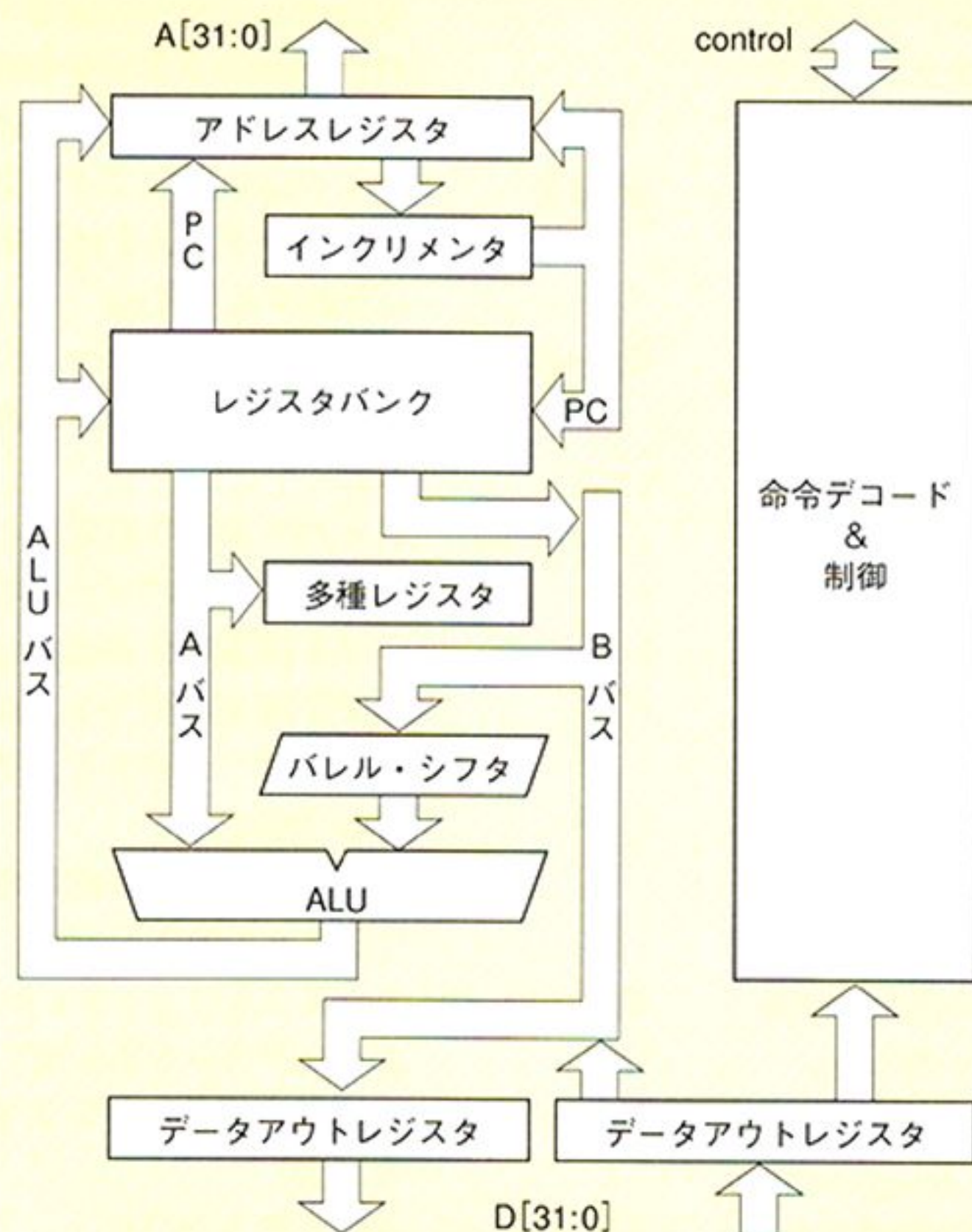
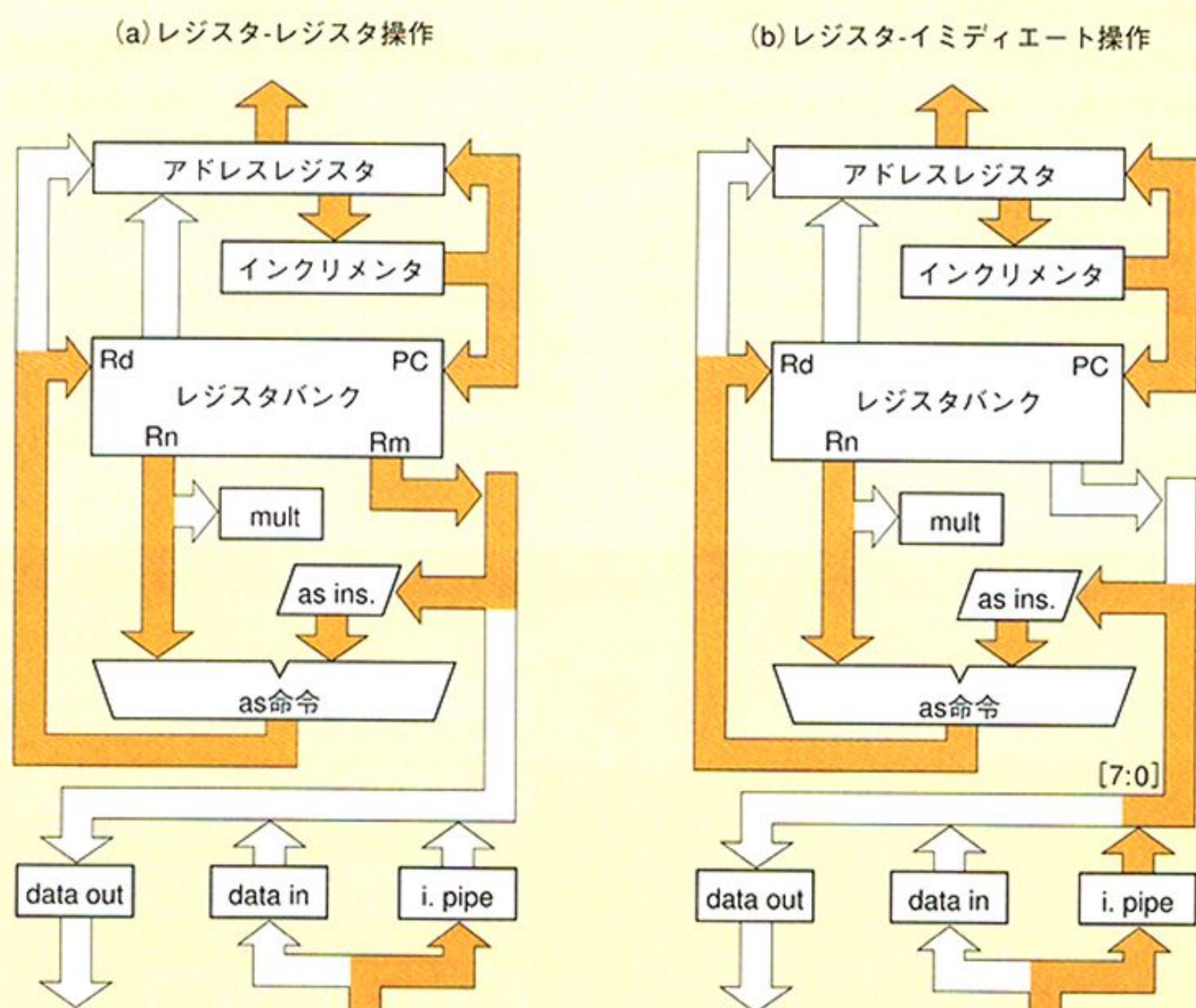



図19　ARMのパイプライン

レジスターレジスタ間演算	fetch	decode	execute		
ストア命令	fetch	decode	calc. addr.	data xfer	
ロード命令	fetch	decode	calc. addr.	data xfer	reg. write
分岐命令	fetch	decode	execute1	execute2	

図20　データ処理命令のデータパスの流れ

えたデータキャッシュアクセスが立ち下がり同期に変更されていることもあり，スーパーパイプライン構造も採用していないので，本当にこんな高周波数で動作可能かは不明である。

MIPSの発表によるとJadeとOpalの性能(MIPS/MHz)はどちらも，Dhrystone MIPSで1.2であるという。これはR3000とほぼ同じ性能である。Opalに関しては，パイプラインのステージ数が増えているのに，Jadeと同じ性能というのは納得がいかない。それは置いておくとしても，同じ性能のIPコアが2つも必要なのかという疑問が残る。MIPSの弁明では，Dhrystoneベンチマークでは真の性能はわからない，実際のアプリケーションではOpalはJadeの2倍の性能があるという。これはキャッシュ容量を2倍にできる点と，64ビット演算と32ビット演算の差と説明されている(嘘っぽいなあ)。

JadeにしろOpalにしろ，論理合成可能なRTL(Register Transfer Level)記述で提供されるのだが，目標動作周波数が達成できるか否かは，LSI製造メーカーの技術力によると思う。

(5)SH1/SH2/SH3，そしてSH5

SHシリーズは日立製作所が1992年に発売した，組み込み用途を狙ったRISC型32ビットマイクロコントローラとして誕生した。その後，積和演算やMMUを内蔵し，MPUとしての地位を確実にしている。多くのRISCが32ビットの固定長命令であるのに対して，SHシリーズは16ビット固定長命令を採用しコードサイズの削減を図っている。

SHシリーズはDLX(R3000)のパイプラインを参考にしているといわれるが，命令によってパイプラインのステージ数が異なっている点でCISCの考え方を引きずっているようにも思える。パイプラインは，次の5ステージから構成される。ステージ構成だけを見ればR3000と同一である。また，遅延分岐は採用しているが，遅延ロードは採用せずデータハザードが生じる場合はインタロックする。

IF：命令フェッチ	メモリから命令を取り込む
ID：命令デコード	取り込んだ命令をデコードする
EX：命令実行	デコード結果に従い，データ演算やアドレス計算を行う
MA：メモリアクセス	メモリのデータアクセスを行う。ロード/ストア命令で発生する
WB：ライトバック	メモリアクセスした結果(データ)をレジスタにライトする

IF，ID，EXの3ステージはすべての命令に存在するが，命令によっては，MA，WBステージがない場合もある。主なパイプラインを図17に示す。図を見ればわかるが，パイプラインはSH1/

SH2とSH3で少し異なっている。レジスターレジスタ間演算（転送を含む）は，SH1/SH2ではIF，ID，EXの3ステージで構成されるが，SH3ではデータを保持するだけのmaステージと，レジスタへ値をライトするためのWBステージが追加されて5ステージ構成になっている。これらの命令において，SH1/SH2/SH3とも，レジスタのリードはEXステージで行っているようである。そして，演算を行った結果は，SH1/SH2ではEXステージのうちに，SH3ではWBステージでレジスタにライトするようである。R3000のパイプラインを参考にした（といわれる）割には，EXステージまでレジスタリードを遅延させたり，SH1/SH2では演算結果をEXステージでレジスタにライトするなど，タイミング的に厳しい設計になっている。まあ，これは，レジスタのフォワーディングをまったく行っていないか，フォワーディングの論理を軽くするためと推測される。もっとも，最新のSH4ではIDステージでレジスタをリードするようになったようで，試行錯誤の痕跡が認められる。

さて，ロード命令はパイプラインの5ステージすべてを使う。WBステージは，最初は，ロードしたデータをレジスタにライトするためだけに存在していたようだ。SH3ではレジスターレジスタ演算にも適用された。一方，ストア命令はレジスタへのライトがないのでWBステージが存在しない。いずれにせよ，命令の種類に応じてパイプラインのステージ数を可変にするのはCISCの発想である。実質的にはパイプラインのスループットは最大のステージ数に支配されるのであまり効果はない。SH3ではそのことに気づいてか，パイプラインがほとんどの命令で5ステージ固定に改善されたが，ストア命令がなぜ4ステージのままなのかは謎である。

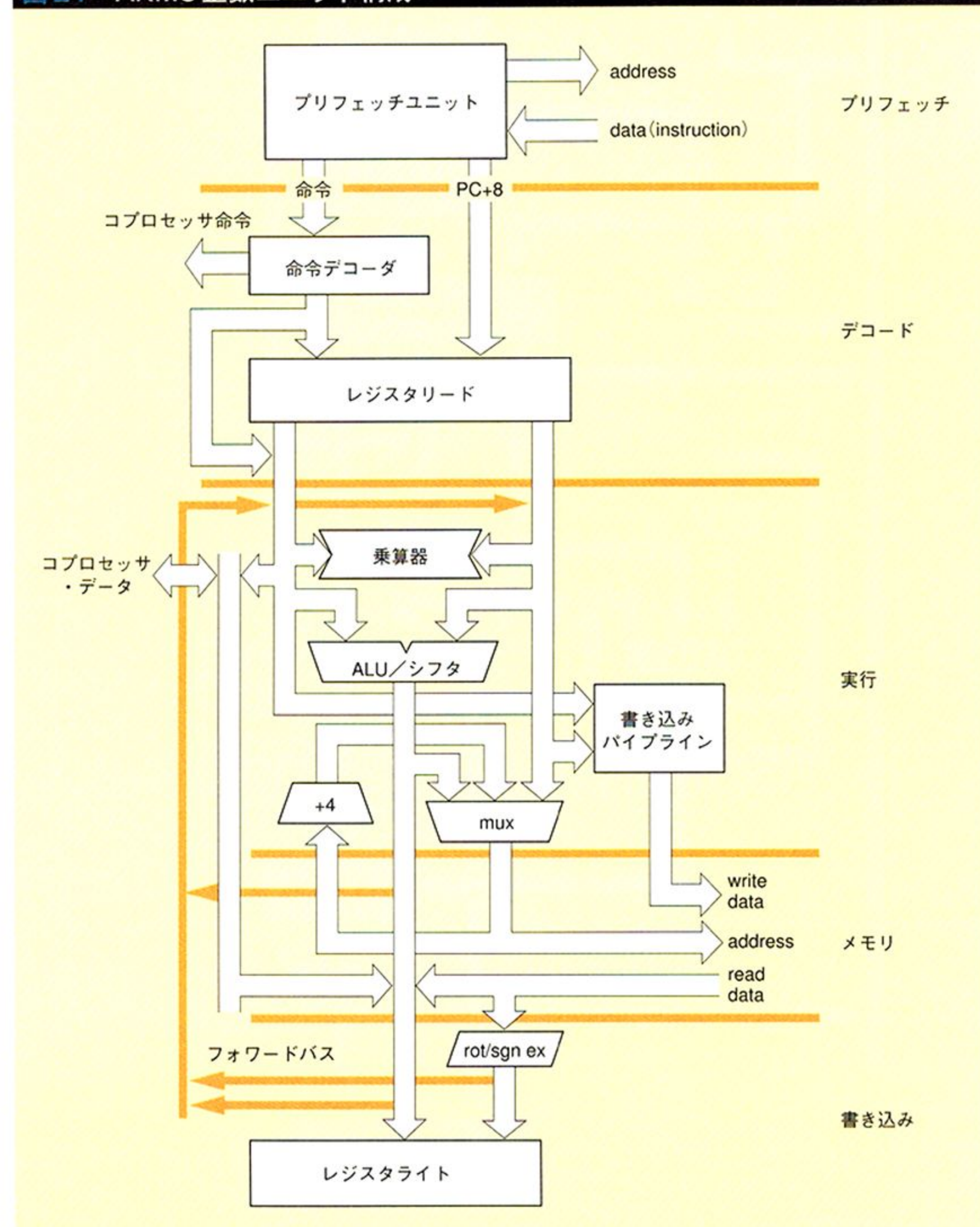
SH4では性能向上のためにスーパースカラ構成をとったが，つい先日発表されたSH5ではシングルパイプラインに戻された。400MHz動作を達成するためには，スーパースカラの制御の複雑さがスピードネックになるという理由からだ。SH5のパイプラインはFetch-1（F1），Fetch-Decode（FD），Decode（D），Execute-1（E1），Execute-2（E2），Execute-3（E3），Writeback（W）の7ステージで構成される。E1，E2，E3ステージからDステージへのフォワーディングが可能という。パイプラインのステージ数の増加に伴う分岐命令の性能低下を補うため，SH5ではSplit Branch（分割分岐とでも訳すのかな）という方式を採用している。これは，分岐先アドレスを計算して命令プリフェッチを行っておき，実際の分岐命令でその命令を実行するという2段階の構造で分岐命令処理を実現する。そのためにPTA（Prepare Target Address）という命令が用意された。

(6)ARM/StrongARM

最初のARMアーキテクチャのMPUが開発された当時，RISCはスタンフォード大学のMIPSと，カリフォルニア大学バークレー校のRISC I，II（Sparcの母体）しか例がなかった。ARMはバークレーRISCを参考にして設計された。ロード/ストアアーキテクチャ，32ビット固定長命令，3オペランドフォーマットという特徴を取り入れたが，レジスタウィンドウ，遅延分岐，全命令の1クロック実行は採用しなかった（ほとんどの命令は1クロックで実行するが）。設計目標はCISCライクな命令セットをRISCに準じた単純なハードウェアで実行することに置いている。

ARMにはARM1～7，ARM8，StrongARMとアーキテクチャに若干の差異がある。それぞれのパイプラインを簡単に見ていこう。まずはARM1～7である。図18にARM7までのMPUのブロック図を示す。ARM7までのパイプラインの基本は単純な3ステージ構成のパイプラインを採用する。ただし，ロード/ストア/分岐命令などはマルチサイクル命令として別のパイプライン処理を行う。それを図19に示す。各パイプラインステージの意味は次のとおり。

fetch	命令フェッチ：メモリから命令をフェッチし，パイプラインに投入する
decode	命令デコード：命令をデコードし，データパスの制御信号を生成する
execute	実行：命令のデコード結果に従い，レジスタファイルをリードし，オペランドを（必要なら）シフトし，演算を行い，結果をレジスタにライトする
calc. addr	アドレス計算：ロード/ストア/分岐命令の場合は演算器を用いてオペランドまたは分岐先のアドレスを計算する。ストア命令ではさらにメモリにストアするレジスタをリードする
data xfer	データ転送：ロード命令ではメモリからオペランドデータをリード

図21　ARM8整数ユニット構成

する。ストア命令ではレジスタの値をメモリにライトする

reg. write　レジスタライト：ロード命令でメモリからリードしたデータをレジスタにライトする

図20にレジスターレジスタ（イミディエート）命令実行時のデータパスのデータの流れを示す。

さて，ARM8ではパイプライン構成が変わった。パイプラインへの命令供給のバンド幅を向上させるため命令のプリフェッチを行いバッファリングする。初代のARM8のプリフェッチユニットには静的な分岐予測機能も内蔵されていたという。図21にARM8のブロック構成を示す。パイプラインは次の5ステージから構成される。

1）命令プリフェッチ
2）命令デコード，レジスタリード
3）実行（シフトと演算）
4）メモリアクセス
5）ライトバック

資料によると，命令の種類によってパイプラインのステージ数が可変になるという記述はない。上に示した5ステージ固定のパイプライン処理を行うと推測される。これでよりRISCらしいパイプラインになったといえる。ARMのパイプラインはARM社とDEC社が共同開発したStrongARM（現在はIntel社に買収されている）で一応の完成を見る。キャッシュの構成が命令とデータに分割された（命令とオペランドフェッチで待ち合わせが生じない）こととレジスタのフォワーディング機能が追加されたのが特筆すべき特徴である。パイプラインは次の5ステージで構成される。

1）命令フェッチ（命令キャッシュから）
2）命令デコードとレジスタリード，分岐先のアドレス計算
3）オペランドのアドレス計算，またはシフトおよび演算を実行
4）データキャッシュへのアクセス
5）レジスタファイルへ結果をライトバック

図22にStrongARMのパイプライン構成図を示す。ARMのパイプラインも命令ごとに可変なステージ数から始まり，結果として5ステージに落ち着いたようである。やはり，5ステージというのがRISCのパイプラインの王道といえるのかもしれない。少なくともこれまでは。

しかし，次期StrongARM（一般にSA2と呼ばれる）では600MHzという高い動作周波数を実現するため，再びパイプラインの見直しがなされた。結果，整数演算で7ステージ，ロード/ストアで8ステージという構成になった（図23）。

パイプラインが2ステージ増えた理由は，主に2本のクリティカルパス（タイミングネックになる論理経路）対策のためである。ひとつ目はALU演算である。従来のStrongARMでは1クロックで，

シフト→ALU演算→条件コードの生成

を行っていた。これを3ステージに分割して処理する。こうすることにより，命令デコードにも余裕ができた。従来は命令デコードとレジスタアクセスを1クロックで行っていたが，

レジスタアクセス→シフト

のタイミングを従来より遅らせて余裕を持たせている。2つ目はデータキャッシュのアクセスである。データキャッシュは，従来は，

アドレスデコード→キャッシュアクセス→データの整列→ALUへ入力

を1クロックで行っていた。SA2ではデータキャッシュが従来の2倍の32KBになったので一度に動作する回路が多くなりクリティカルパスになった。そこでデータキャッシュアクセスを2クロック（2ステージ）で行うように改良した。

図24にSA2の機能ブロックを示す。

SA2ではパイプラインのステージが増加したため，分岐命令の性能低下（当然，分岐予測機構は備えている）などを考慮するとCPIが5～8%増加するが，周波数を1.5倍に向上できるという。差し引き40%程度の性能向上となる。なお，分岐ターゲットバッファは128エントリからなるダイレクトマップキャッシュで，2ビットの情報で分

図22　StrongARMコア・パイプライン構成

岐の履歴を管理する。

先日，インテルはSA2のマイクロアーキテクチャをXScaleという名称で大々的に発表した。しかし，インテルが公開している資料を見る限り，XScaleの内容は最初にSA2として発表されたものと同じである。目新しいところは，製品系列に800MHz品と1GHz品が追加されたことくらいである。XScaleではSA2のパイプラインはスーパーパイプラインと呼ばれるようになった（実質はなにも変わっていない）。

(7)V810/V830/V850

V800シリーズはNECがV80の後継として開発した，どちらかといえば，マイクロコントローラといえるMPUシリーズである。その開発目標は低価格で低消費電力のチップであった。V800シリーズでは基本の命令長を16ビットとしながらも，大きいビット数のイミディエート値やディスプレースメントの指定でコード効率を上げるために32ビットの命令長も用意している。ちょうどSHとARMの中間のようなアーキテクチャである。V800シリーズは，1992年に最初のV810が開発され，その後V830，V850が続いて開発された。現在は，これらのMPUをCPUコアとした周辺内蔵品が販売されている。

V810のパイプラインに関してはユーザーズマニュアルには記載されていない。しかし，当時の雑誌の解説記事によると，

フェッチ
デコード
実行
書き込み

という典型的（と記述されている）なRISCのパイプラインに対し，デコードと実行の間にレジスタリードのステージを挿入したという。これは可変長の命令フォーマットをデコードするのに長い処理時間が必要であり，デコードに余裕を持たせるためと説明されている。合計5ステージ構成のパイプラインであるが，これは33MHz以上の動作周波数を想定したものであり，25MHzの動作周波数ではレジスタリードのステージは結果として不要だったようだ。しかし，このパイプライン構成ではロード/ストア命令でのオペランドフェッチステージがない。おそらく，ロード/ストア命令の処理時には6ステージになるのであろう。

その後，V830/V850になるとパイプラインの見直しが行われ，

IF	命令フェッチ
RF	命令デコード
EX	実行
MEM	オペランドアクセス
WB	ライトバック

の5ステージのパイプラインになった。これは図5に示した典型的なRISCのパイプラインと同じで特に取り立てていうこともない。ただ1点，分岐命令は，TAKENする場合，EXステージの終了を待ってIFステージを開始する。これは図9(c)と同じである。明らかに分岐先のアドレス計算から命令フェッチまでに時間的余裕を持たせている。このため，分岐命令のレイテンシは3クロック（NO TAKENの場合は1クロック）になる。V800シリーズでは遅延分岐を採用しない（分岐予測もない）ため，分岐が多いプログラムの処理は不利になる。また，歴史的には古いMPUのせいか分岐予測機構も採用していない。シンプル・イズ・ベストという考え方なのだろう。

●おわりに

シングルパイプラインの概要について説明してきた。思えばヘネパタは偉大だった。いうまでもないがHennessyはMIPS RISCの生みの親，Pattersonはバークレーの RISCの生みの親である。この2人よって著されたヘネパタは日本のRISCメーカーの技術者に多大な影響を与えた。日立のSHシリーズやNECのV800シリーズはヘネパタで示されたアーキテクチャ（特にパイプライン）を参考にしているといわれている。それだけDLX（R3000）のパイプラインが洗練されているということなのであろう。ヘネパタは，日本語訳も出ているので未読の方は一度は目を通すことをおすすめする。さて，次回は2命令以上を同時実行するパイプラインであるスーパースカラに関して解説する。

図23　SA2のパイプライン

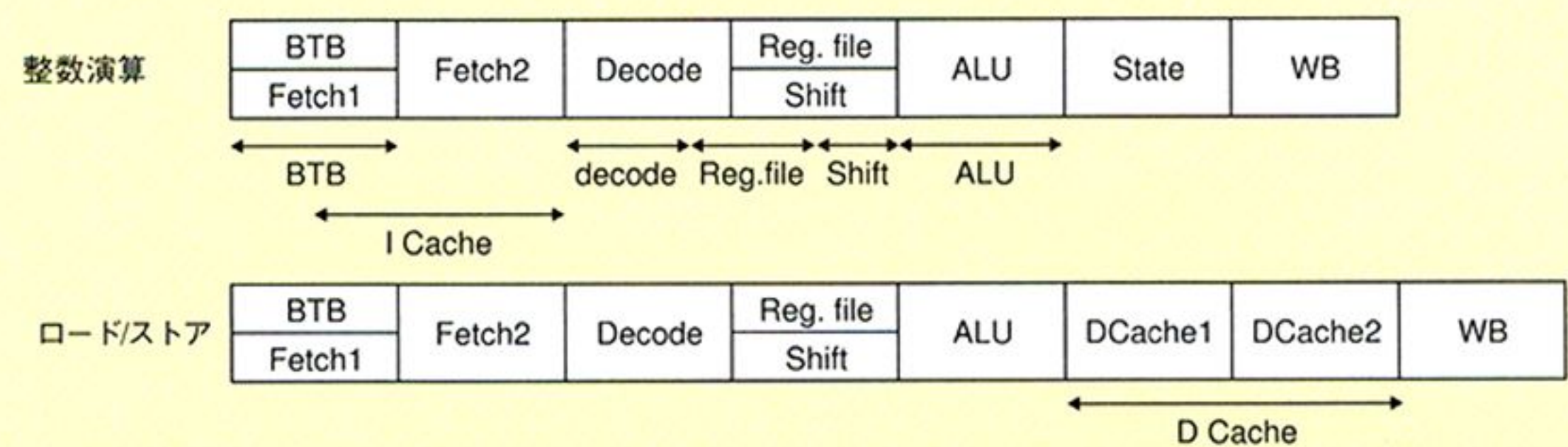
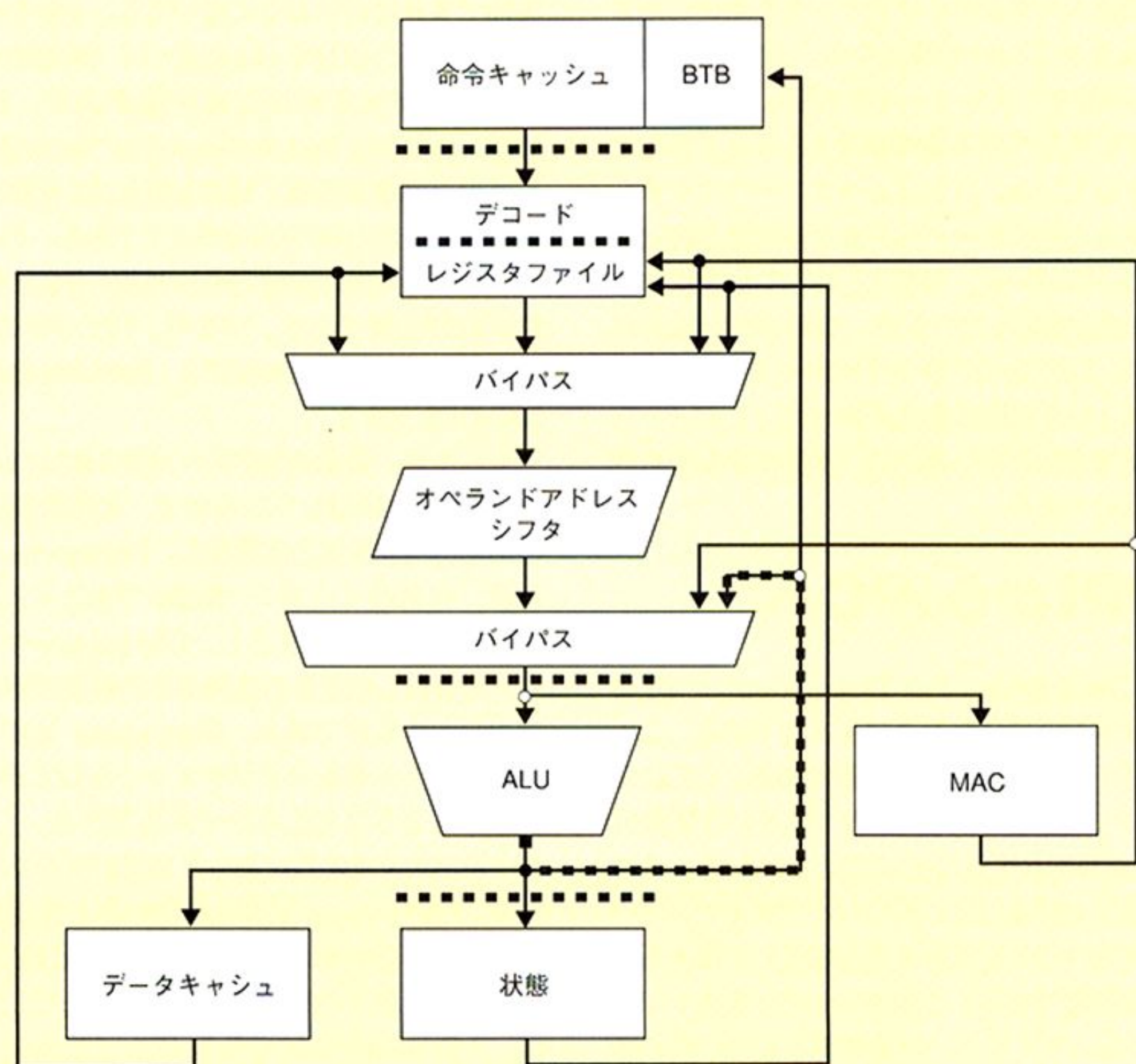



図24　SA2のブロック図

Computer Architecture

コンピュータアーキテクチャ

その直感的アプローチ ④

並列処理の基本，スーパースカラ

中森 章 *Nakamori Akira*

1命令1クロック処理が当たり前になってくると，プロセッサの性能はクロック数（と命令の機能）だけで決まってしまい，アーキテクチャ的には進化の余地はないように思える。そこで登場するのが1クロックで複数の命令を同時に実行してしまおうというアプローチだ。その代表がスーパースカラという手法であり，現在のMPUの多くで採用されている。ここではスーパースカラの基本的な考え方を押さえておこう。

前回はシングルパイプラインについて解説した。今回はシングルパイプラインを多重化したスーパースカラについて解説する。現在の高性能MPUは例外なくスーパースカラ構造を採用し，1クロックに実行できる命令数を1よりも大きくする工夫をしている。コンピュータアーキテクチャを理解するためにスーパースカラの基礎知識は必須である。とはいえ，最近（といってもかなり前だがの雑誌を読んでいると，AMDのAthlonは（発行と完了が）完全アウトオブオーダーなので素晴らしいという意味の記述があった。このような誤った認識を持たないために，本稿が少しでも役立てば幸いである。

●CPIからIPCへ

CPI（Clock cycles Per Instruction）とは1命令を実行するのに必要なクロック数である。この値が小さいほどMPUは高性能である。CISCからRISCへの進化によってCPIが1という限界に達した（理想的な実行環境に限定されるが）。それ以上性能を上げるには，同じパイプラインステージ内で複数の命令を実行させればよいと考えるのが自然な発想である。これがスーパースカラである。そうなってくると，性能指標としてCPIの逆数であるIPC（Instructions Per Clock cycle）を使用したほうがわかりやすい。つまり，1クロックに実行できる命令数である。2命令を並列に実行できればIPCは2に近づくし，4命令を並列に実行できればIPCは4に近づく（理論的には）。IPCは値が大きいほど高性能を表す。IPCはMIPS（Million Instructions Per Second）値とも密接な関係がある。MIPS値とは1秒間に実行できる命令数（100万命令単位）である。その意味で，IPCに動作周波数（MHz単位）を掛け算した値がMIPS値である。つまり，IPCが1の場合，100MHz動作では100MIPS，200MHz動作なら200MIPSである。

もっとも，最近のMPUのMIPS値はDhrystone MIPSを採用しているので，公称性能は本来の意味のMIPS値とは異なる。Dhrystone MIPSとは，有名なミニコン（死語）であるVAX-11/780の性能を1MIPSとし，Dhrystoneベンチマークを実行したときの性能がその何倍の値になるかを示したものである。Dhrystone MIPSを用いれば，シングルパイプラインでもIPCが1を超えているように見えるので多用される。しかし，スーパースカラ構造になると事情が異なる場合もある。Dhrystone MIPSを用いると性能がそれ程高く見えないからだ。その代わり，IPCと同時実行できる命令の数が等しいと仮定して，たとえば，2命令同時実行可能なパイプラインを200MHzで動作させると400MIPS（200MHz×2命令という計算）などという理想値を示す場合もある。現実には，同時実行できる命令数を増やしていくとき，IPCは1.6辺りに収束することが経験的にわかっているので，2命令同時実行でもIPCが2になることはまずない。ただし，Dhrystone MIPSを真のMIPS値と（意図的に）混同してIPCを計算すれば，2命令同時実行で2.2程度になることもある。この場合でも，4命令同時実行では4.0どころか3.0を超えることはまずない。その場合は動作周波数×4でMIPS値が決められたりする。つまり，200MHzで動作し，4命令同時実行なら800MIPSといった具合である。まあ，公称MIPS値をそのまま信じる人はいないと思うが，このような数字のマジックに惑わされないようにしなければならない。しかし，感覚的にはIPCが2.2などといわれると非常に高性能と思ってしまう。現在のGHz単位で動作するx86系のMPUのIPCは2～3などといわれているが，Dhrystone MIPSによるIPCでは0.6程度である（つまり実質的な性能は，動作周波数の割には高くない）。

一般にパイプラインのステージ数を増加するとIPCは低下する。動作周波数を向上させるためにパイプラインのステージ数を増すのはよくある手法だが，パイプラインのステージ数を増加させてもIPCを0.6程度に保ち続けているインテルやAMDは称賛に値する。NetNews（fj.comp.arch）にPentium Ⅲ-750MHzでDhrystoneベンチマークを行った場合の性能の実測値が報告されていた（メッセージIDは失念した）。その値から計算すると，真のIPCは1.01，Dhrystone MIPSによるIPCが1.18であった。予想の2倍の性能になっているが，これはDhrystoneという，最高性能を発揮しやすい，プログラムの性質によるものだろう。実際のアプリケーションではこうはいくまい。ちなみに，別の資料によるPentium（P5）-66MHzのDhrystoneによるIPCは1.5なので，Pentium ⅢになるとIPCは低下している。パイプラインのステージ数が増加しているので，当然といえば当然か。

とにかく，シングルパイプラインの目標がCPIを1に近づけることであったように，スーパースカラの目標はIPCを同時実行できる命令数に近づけることである。まあ，x86は独自の道を歩んでいるようにも思えるが。

●スーパースカラの概念

複数の命令を並列実行する機構をスーパースカラ（superscalar）と呼ぶが，名前の由来に触れておこう。スカラから連想されるのは，ベクトル量に対するスカラ量である。つまり，科学技術計算でお馴染みのベクトルや行列演算に特化した並列処理ではなく，スカラ量に対する並列処理という意味でスーパースカラと呼ぶという説が有力である。この意味で通常のシングルパイプラインをスカラパイプラインと呼ぶこともある。また，スーパースケーラという呼び方もある。これは，1クロックで1命令を実行するという直感的な基準（スケーラ）を超えるという意味からきているらしい。この説はあまり聞いたことはないが，技術解

説で有名な某誌ではそう説明されている。とどのつまり，スーパースカラの語源ははっきりしない。ただ，最近の論文ではスーパースカラの反意語としてユニスカラ（uniscalar）が使用されるが，これなどは「スカラ＝パイプラインの本数」という概念からであろう。スカラかスケーラかというのは，個人的には単なる発音の問題だと思う。英語による発音はスーパースケーラに近い（少なくともあのHennessy教授はそう発音していた）のだが，最近ではスーパースカラと表記されるほうが多いように思う。実際にスーパースカラと発音する外国人も多くなった。

スーパースカラでは並列に実行できる命令数をウェイと呼ぶ。イシュー（issue：発行）と呼ぶ場合もある。厳密には命令デコーダから複数存在する命令実行パイプラインに同時に送り込む（発行）ことのできる命令数がイシューであり，命令実行パイプラインの本数がウェイである。しかし，現在ではそれほど厳密には区別されていない。どちらかといえばウェイという表現のほうがよく使われる。一般に，2ウェイスーパースカラといえば2命令を並列実行できるパイプライン構造のことである。しかし，アウトオブオーダー実行が当然のようになっている現在の技術では，複数存在する演算器に対して2命令を同時発行できるパイプライン構造（2イシュー）のイメージのほうが強い。いずれにしろ，スーパースカラの概念は図1のようなパイプラインの図で表されることが多い。つまり，命令フェッチ，命令デコード，実行，メモリアクセス，ライトバックを2命令並行に処理する，という感じである。実際の動作とはあまり一致していないが直感的ではある。

連続する命令は互いに独立ではなく，相互に関係がある場合がある。このため，単純に命令の並列実行はできない。因果律が逆転するからだ。スーパースカラの最大の特徴は，MPUが複数の命令を並列に実行するからといって，プログラムで特別な考慮をする必要がない点である。従来からの命令セットを変更する必要もない。MPU自身が命令間の依存性を検出し，並列に実行可能な命令を自動的に判定し，演算器に対して発行する。そして各演算器は命令を並列に実行する。もっとも，命令間に依存関係があると，処理にオーバーヘッドが生じ，実行効率が低下するので，スーパースカラの真の性能を発揮するには，プログラムでの考慮（コンパイラによる命令の並べ替え）が必要である。このため，新しいMPUが発表になるとき，従来のオブジェクトコードそのものではそこそこしか速くならないが，新しいMPU用に開発されたコンパイラで再コンパイルすると性能が劇的に向上する，ということがよくいわれる。

●スーパースカラの実現

一般に，命令はデコーダでの発行，演算器での実行，実行の完了の過程を経て処理される。命令の発行はプログラムに書かれた順序で行うこともできるし，矛盾を生じない限りは，プログラムの順序を無視して行うこともできる。また，命令の実行は基本的に1サイクルなので，通常は発行された順序で完了する。ただし，実行クロック数の異なる命令を同時に発行すると，完了する順序が入れ替わることもある。当然，プログラムの順序で完了するとは限らない。処理がプログラムの順序どおりであることをインオーダー，プログラムの順序と異なることをアウトオブオーダーと呼ぶ。スーパースカラの方式は，プログラムの発行，完了が，それぞれ，インオーダーかアウトオブオーダーであるかによって4種類に分類できる。以下は簡略化のために2ウェイのスーパースカラを想定して説明する。

0）命令デコード

命令デコードはインオーダー/アウトオブオーダーでそれほど大きな違いはない。命令キャッシュから2命令（たいていの場合ウェイの数と同じ数）をデコードし，命令キューに入れておしまいである。デコーダと演算器の中間に命令キューを持つ方式では，デコードと発行を独立に行えるので，（命令キューに空きがある限り）各サイクルごとに2命令をデコードできるので効率がよい。また，逆に考えると，命令キャッシュの参照が少々もたついても，その時間的なロスを命令キューで吸収して見えなくすることが可能である。事実，MIPSのRuby（20Kc）はウェイ予測を行って命令キャッシュを参照しているが，予測失敗時のペナルティは命令キューで吸収できると説明している。

命令キューの役割は，デコードと命令実行開始までの待ち時間を最小にする役割もあるが，特にスーパースカラにおいては命令間のオペランドの依存関係を調べることである。たとえば，片方の（先行する）命令の実行結果を，もう片方の（後続する）命令がソースオペランドとして使用する場合に依存関係があるという。簡単にいうとレジスタ間のハザードである。もし，2命令間に依存関係がなければ，同時に実行可能なので，演算器に2命令（ウェイの数）を発行する。命令の追い越しを許さない場合（つまり，インオーダー）は，デコードしている2命令間でのみ依存性を調べればよいので，いちいち命令をキューに入れなくてもデコーダのみの検査でこと足りる。このためインオーダーなスーパースカラ構造では命令キューを持たないものも多い。ただし，依存関係がある場合はデコーダで（依存関係が解消するのを）待ち合わせることになるので，各サイクルごとに常に2命令をデコードすることはできない。このため，少し効率が悪い。

なお，このような役割をする命令キューは特別にリザベーションステーション（Reservation Station）とか集中命令ウィンドウ（Central Instruction Window）とか呼ばれる。

また，命令の依存関係は命令キューで解消されているので，いったん演算器で実行が開始されたらレジスタ間のハザードによるストールは発生しない。命令ごとに定められた実行クロック数（レイテンシ）を経て実行が完了する。ただし，データキャッシュアクセスによるストールは発生する可能性がある。多くの場合，データキャッシュにアクセスするためのロード/ストアユニットは実行ユニットとは分離されているので，アウトオブオーダーの場合は他の命令実行に影響を与えない。インオーダーの場合は後続命令が先行するロード/ストア命令を追い越せないので，データキャッシュにアクセス中はパイプラインがストールしてしまう。

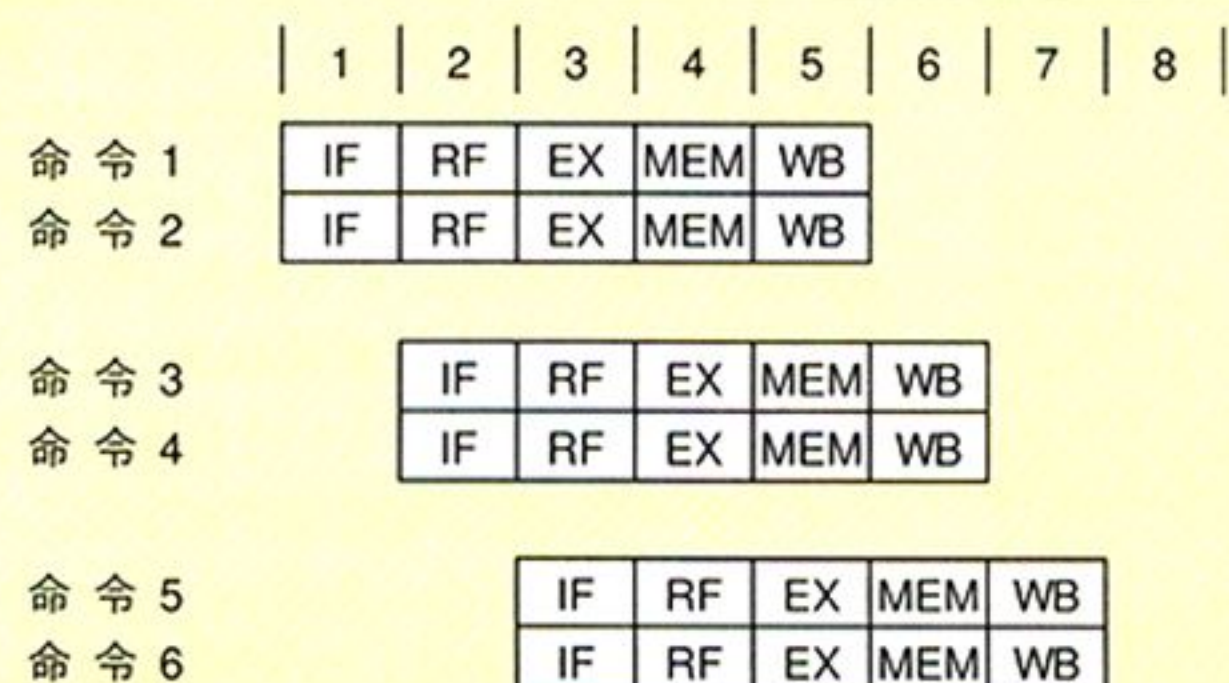
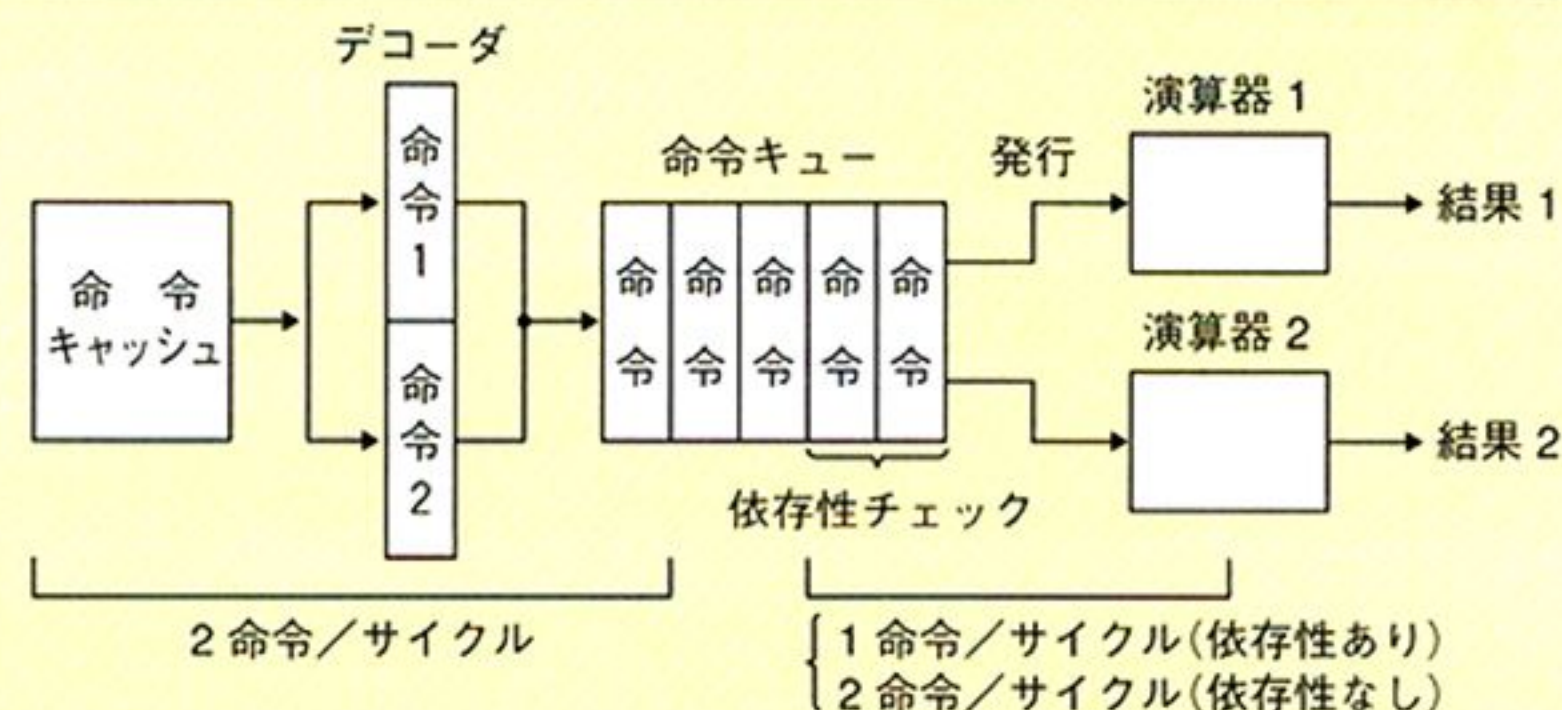
1）インオーダー発行

この場合は，命令キューの先頭の（または現在デコードしている）2命令（ウェイの数）のみの依存性を調べる。命令間に依存性がない限り，2命令（ウェイの数）を同時に発行する。依存性がある場合は1命令のみを発行する。残った命令はその次の命令と組になり，再び依存関係が調べられ

図1　スーパースカラ（2ウェイ）の概念図

	1	2	3	4	5	6	7	8
命令1	IF	RF	EX	MEM	WB			
命令2	IF	RF	EX	MEM	WB			
命令3		IF	RF	EX	MEM	WB		
命令4		IF	RF	EX	MEM	WB		
命令5			IF	RF	EX	MEM	WB	
命令6			IF	RF	EX	MEM	WB	

図2　インオーダー発行（2ウェイ）

デコーダ
命令キャッシュ
命令1
命令2
命令キュー
命令 命令 命令 命令 命令
発行
演算器1
演算器2
結果1
結果2
依存性チェック
2命令／サイクル
1命令／サイクル（依存性あり）
2命令／サイクル（依存性なし）

る（図2）。

2）アウトオブオーダー発行

この場合は命令キュー全体（あるいはある決められた命令数の間）で依存性が調べられる。オペランドの依存性がない（というか，オペランドをすぐに利用できる）命令のうち，先頭から2命令（ウェイの数）を同時に発行する。発行される順序はプログラムの順序と入れ替わる場合がある。命令キュー内のすべての命令になんらかの依存関係がある場合は，先頭の1命令のみが発行される（図3）。

3）アウトオブオーダー完了

RISCは基本的には命令を1クロックで処理できるが，現実には実行に数クロックかかる命令も存在する。特に浮動小数点命令は実行に最低でも3クロック程度かかるのが実情である。つまり，MPU内には複数の演算器が存在するが，それらが処理する命令のレイテンシは一般には異なる。ということは，命令がインオーダーに発行されようがアウトオブオーダーに発行されようが，実行がプログラムの順序で完了する保証はどこにもない。すなわち，スーパースカラではアウトオブオーダー完了が自然な姿である（図4）。また，実行が終わった演算器には命令キューから次々と命令を発行すればいいので効率的でもある。

しかし，実行の完了と同時に結果をレジスタファイルに書き戻していたら不都合が生じる場合がある。

まず，第1は出力依存関係である。同時に実行されている命令のデスティネーションレジスタ（結果の格納先）が等しい場合，そこにはプログラム的に後ろにある命令の実行結果が書き込まれなければならない。ところが，後続命令の処理が先に完了し，先行命令の処理があとから完了する場合，正しい結果（後続命令の結果）が破壊されてしまう。これがWAW（Write After Write）ハザードである。シングルパイプラインではWAWハザードは起こりえないがスーパースカラでは当たり前に発生する。もっとも，これはあとで説明するレジスタリネーミングで回避することができる。

しかし，インオーダー発行のスーパースカラでは（回路規模が増大するのを嫌って）レジスタリネーミングを行わないことも多く，この場合はデコード時に発行を待ち合わせるとか，後続命令のライトをストールさせるとか，なんらかの対策が必要である。

第2は，こちらのほうがもっと深刻であるが，例外の正確性（precise）の問題である。例外はプログラムの順序で処理されなければならない。たとえば，先行する命令も後続する命令も例外を発生する場合，後続命令の例外が先に検出されても，先行命令の例外発生を優先させるようにしなければプログラム処理に矛盾が生じる。

しかし，例外をプログラムの順序で発生させるという制約はシングルパイプラインでも同様であり，通常なんらかの対策が施されている。問題は例外（割り込みでも同様）発生後に，例外の発生直後の命令からプログラムの処理を再開させる場合（ブレークポイント命令，システムコール命令，トラップ命令，割り込みなどの処理）に生じる。つまり，レジスタへのライトがプログラムの順番を無視して行われていたら，例外からの再開をどの命令から開始してよいのか判断できない。

たとえば，op1，op2を適当な演算として，

```
R1 <- R2 op1 R3   …  命令1
R3 <- R4 op2 R5   …  命令2
例外/割り込み      …  命令3
```

という命令処理を考え，op2の実行がop1よりも早く終了すると仮定する。この2命令の処理中に後続命令で検知される例外とか割り込みが発生すると，R1は更新されていないのに，R3が更新されている状況が発生する。この場合，プログラムの実行再開は，まだ実行されていない命令1から行うことになる。しかし，命令1のソースオペランドであるR3は命令2で更新される前の値が必要なので矛盾が生じる。例外の正確性を維持するのは，シングルパイプラインではそれ程複雑な制御ではないが，（アウトオブオーダー完了の）スーパースカラではかなり複雑である。

例外が発生すると，それは致命的とみなし，プログラムの実行を中断する（再開しない），割り込みの受け付けは再開に都合のいい時点の処理が終了するまで待ち合わせる，という制御を行えば，アウトオブオーダー完了を実現できる。ただし，システムコールが行えないとか，割り込み応答性が悪くなるという問題が生じ，あまり現実的ではない。

4）インオーダー完了

インオーダー完了とは，各演算器の完了がアウトオブオーダーに完了するのは避けられないので，その結果を，いったん別の場所に保存しておき，レジスタにライトする順番をプログラムの順番に一致させる方式である。レジスタへのライトが終了するときに初めて命令は真の完了となる。この場合，演算器での実行完了と，命令の真の完了を区別する必要がある。一般には，前者をコンプリート（complete），後者をリタイアメント（retirement）と呼ぶ。リタイアメントはコミット（commit）と呼ばれることもある。なお，本稿ではリタイアメントという表現は長ったらしいので，その動詞形のリタイアという表現を使用する。

インオーダー完了を実現するために，リオーダーバッファ（Reorder Buffer：並べ替えバッファ＝ROBと略される）という機構が導入される。リオーダーバッファとはプログラムの実行順序を

図3　アウトオブオーダー発行（2ウェイ）

図4　アウトオブオーダー完了（2ウェイ）

	1	2	3	4	5	6
演算器1	EX	EX	EX	EX	EX	WB
演算器2	EX	EX	WB			

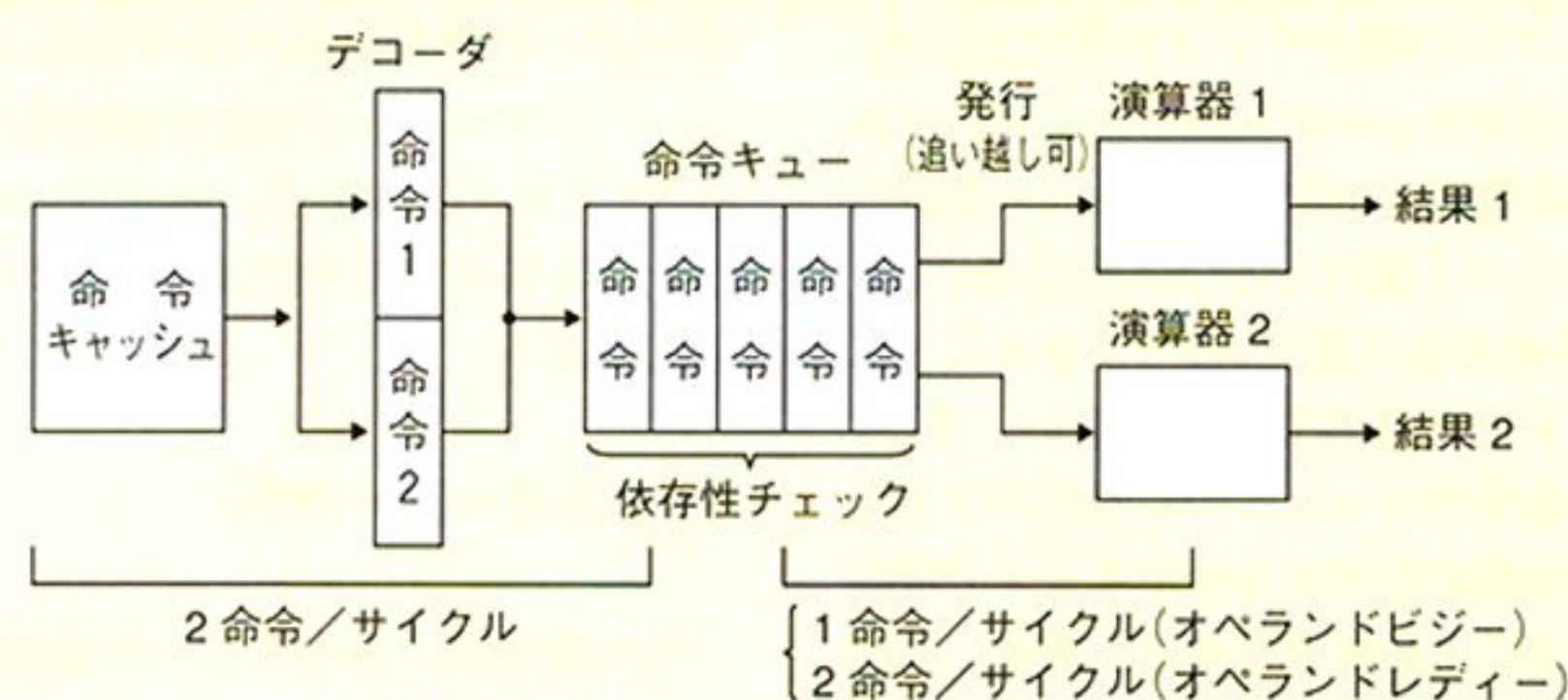
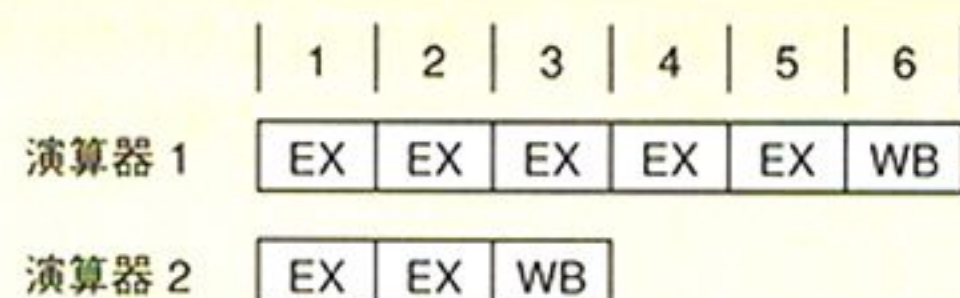
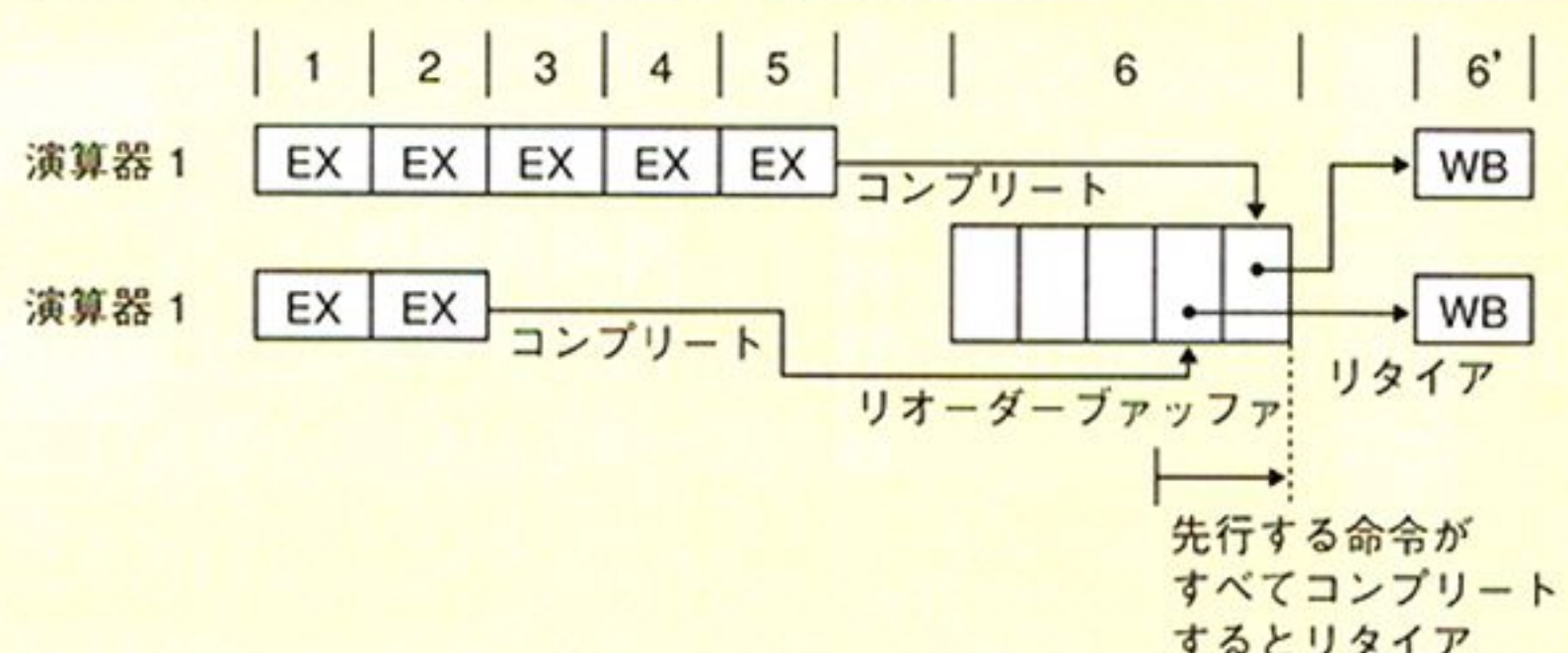



図5　インオーダーオーダー完了（2ウェイ）

記憶しておくテーブルで，命令の発行時に適当な情報が設定される。その各エントリは，命令がコンプリートしたか否かの情報，命令の実行結果を一時退避するバッファ（このバッファはROBにない場合もある）などからなる。

ROB内にある命令の先頭から，連続してコンプリートしている命令がリタイアできる（図5）。1サイクルにリタイアできる最大命令数はMPUごとに異なるが，多くの場合，スーパースカラのウェイ数に等しい。たとえば，命令1から命令4がプログラムの順序であるとき，ROBの内容が，

```
命令1　未コンプリート
命令2　コンプリート
命令3　コンプリート
命令4　…
```

となっている場合，このサイクルでは1命令もリタイアできない。一方，

```
命令1　コンプリート
命令2　未コンプリート
命令3　コンプリート
命令4　…
```

となっている場合は，このサイクルでは命令1のみがリタイアできる。また，

```
命令1　コンプリート
命令2　コンプリート
命令3　コンプリート
命令4　未コンプリート
```

となっている場合は，命令1，命令2，命令3がリタイア対象である。ただし，実際にリタイアできる命令の最大数はMPUごとに異なる。もし，MPUが1サイクルで2命令がリタイア可能なら，命令1，命令2のみがリタイアし，命令3は次のサイクルでのリタイアに回される。もし一度に4命令がリタイア可能なら，命令1，命令2，命令3のすべてがリタイアできる。

例外の正確性の問題があるので，特殊な場合を除き，アウトオブオーダー完了という仕組みは採用されない。したがって，スーパースカラの種類は，実質的には，インオーダー（インオーダー発行，インオーダー完了）とアウトオブオーダー（アウトオブオーダー発行，インオーダー完了）の2種類しかない。図6に典型的なスーパースカラ構成のMPUのブロック図を示す。図6(a)ではリザベーションステーションはひとつのみであるが，図6(b)のように（いくつかの）演算器ごとにリザベーションステーションを設ける構成もある。

また，インオーダーなスーパースカラは2ウェイのものが主流である。インオーダーで3～4ウェイというのは記憶にない。これは，多ウェイのスーパースカラ構造を採用する場合でも，整数ALUは2個程度しか用意されていないためではないだろうか。インオーダーなスーパースカラではウェイの数だけALUがないとパイプライン効率が悪い。それなら，いっそアウトオブオーダーにしたほうが同時発行の効率が上がる。

●レジスタリネーミング

レジスタリネーミング（Register Renaming）とはその名称のとおりレジスタ名の付け替えである。その役割には2つある。基本的にはスーパースカラの命令発行を効率的に行うための技術である。

第1は，アーキテクチャ的に定義されたレジスタ数を増やすことである。たとえば，x86系のMPUの汎用レジスタは8本しかないので，ちょっとしたプログラムでもレジスタの使い回しが多くなり，レジスタの依存関係が発生しやすい。これは命令発行の制約となる。レジスタの本数をアーキテクチャが規定するより大きく持ち，レジスタの名前を付け替えることで，内部的に，プログラムの依存関係を低減できる。

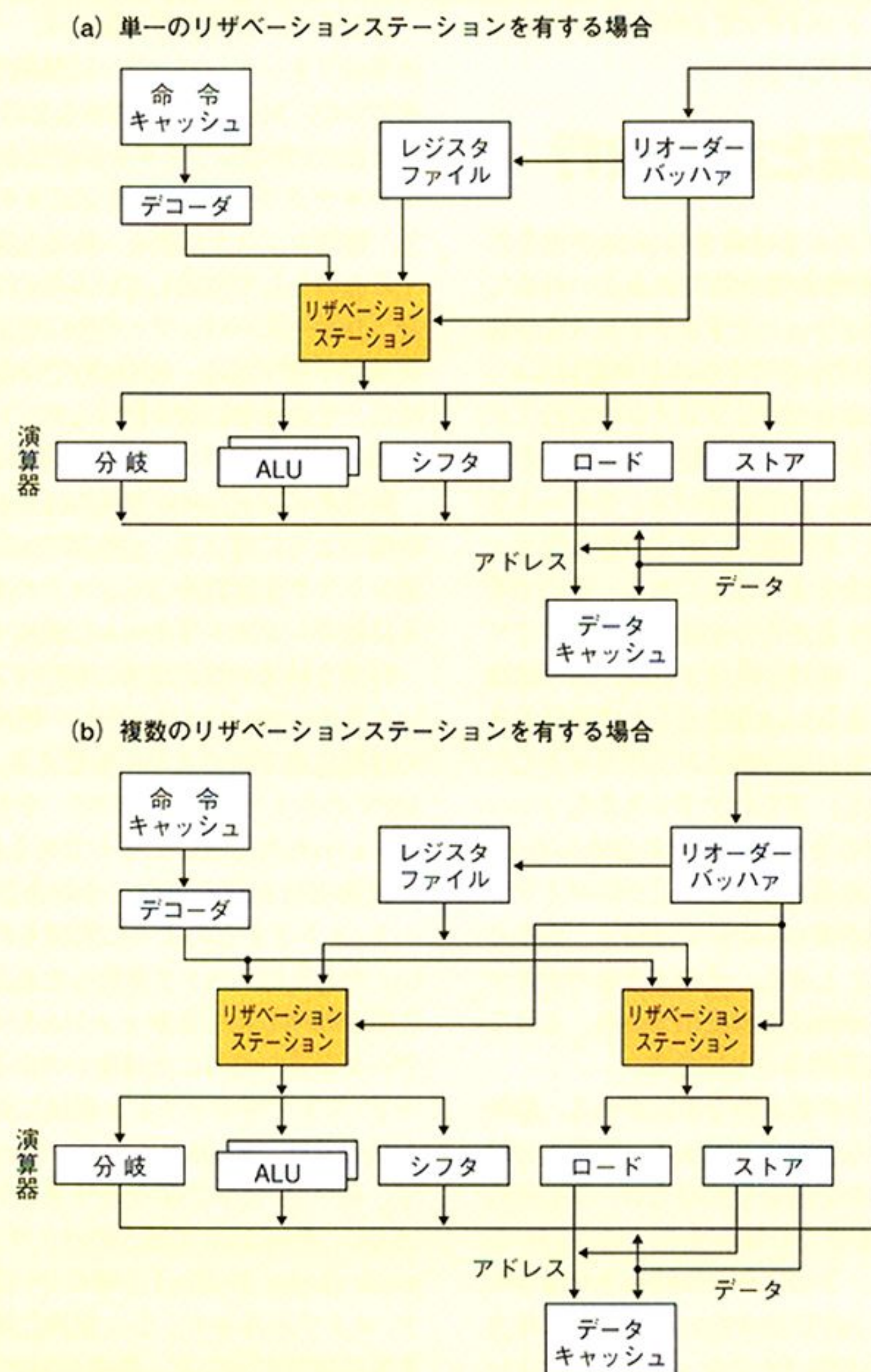
第2は，これも依存関係の解消であるが，WAR（Write After Read）ハザード，WAW（Write After Write）ハザードという偽の依存関係を解消することである。偽の依存関係とは，本来はハザードになるが，レジスタリネーミングによって依存性を解消でき，結果として命令発行の妨げとならないように変更可能な依存関係である。なお，WARとは先行する命令のソースオペランドを後続の命令で変更する可能性のある依存関係，WAWとは後続命令が変更したデスティネーションレジスタを先行する命令が変更する可能性のある依存関係である。これらは同一のレジスタが同時に変更される場合に生じるので，その同一のレジスタを別々のレジスタに割り当ててやれば（偽の）依存関係がなくなる。

また，真の依存関係とはRAW（Read After Write）ハザードのことであり，後続命令が先行する命令の実行結果をソースオペランドとして利用する場合である。これはレジスタリネーミングによっても解消できない。たとえば，次のような命令列を考える。opは単純な加算（+）よりもレイテンシの大きい演算とする。

```
R3 <- R3 op R5 … 命令1
R4 <- R3 + 1   … 命令2
R3 <- R5 + 1   … 命令3
R7 <- R3 op R4 … 命令4
```

図6　典型的なスーパースカラ構成

(a) 単一のリザベーションステーションを有する場合

(b) 複数のリザベーションステーションを有する場合

この4命令は（4ウェイスーパースカラで）同時発行しようとしても，命令2と命令3がWARハザードに，命令1と命令3がWAWハザードになっているため，同時発行できない。そこで，次のようにデスティネーションレジスタに対してレジスタリネーミングを行う。基本的にはデスティネーションレジスタを別個のレジスタに割り当てればよい。ソースオペランドはデスティネーションレジスタの割り当てにしたがって適宜変更される。

```
P1 <- R3 op R5    …  命令1
P2 <- P1 + 1      …  命令2
P3 <- R5 + 1      …  命令3
P4 <- P3 op P2    …  命令4
```

このとき，WARハザードとWAWハザードは解消される。しかし，命令1と命令2，命令3と命令4，命令2と命令4は依然としてRAWハザードの関係にある。この場合，依存性のない命令1と命令3がまずアウトオブオーダー発行できる。命令2は命令1がコンプリートするときに，命令4は命令2と命令3がコンプリートするときにソースオペランドが確定するので，晴れて発行できるようになる。

なお，レジスタリーネミングという表現も長ったらしいので，以下ではその動詞形のレジスタリネームという表現を用いる。

●分岐予測と投機実行

典型的なプログラムでは命令の10％が無条件分岐，10～20％が無条件分岐であるといわれている。無条件分岐はフェッチするアドレスを分岐先に切り替えるだけなのでそれほど問題はない。一方，条件分岐は命令がパイプラインの実行ステージでコンプリートするまで分岐するか否かが不明なので厄介である。分岐命令のコンプリートを待っていたのでは，その間に，多くの命令をフェッチし発行する機会を失うことになる。そこで考案されたのが分岐するか否かを推測するアルゴリズムである。もし，推測が成功すれば，命令はほんの少しの遅延（あるいは遅延なし）で続行できる。推測が失敗すれば部分的にコンプリートしている命令を無効化し，正しいアドレスからフェッチ，デコード，発行を再開しなければならない。これは，最近のx86系のMPUのようにパイプラインのステージ数が多いMPUでは特に，かなりの性能低下である。しかし，そのようなペナルティを考慮しても，分岐予測は必須であり，これを行わないと性能は悲惨なことになる。

分岐予測には2つの基本的な手法がある。静的な分岐予測と動的な分岐予測である。静的な分岐予測はコンパイラが分岐命令の命令コードに埋め込んだ「ヒント」情報で分岐が発生するか否かを予測する。ただし，このような分岐命令を命令セットとして有するMPUは少ない。あるいは後方（backward）への分岐（オフセットが負）はループの終端と見なせるので，これを分岐すると予測するのも静的な分岐予測といえる。静的な分岐予測と動的な分岐予測を比較すると，一般には，動的な分岐予測のほうが効果的といわれている。動的な分岐予測とは分岐命令の時間的な挙動を評価する。一度分岐した分岐命令は次も分岐する傾向があると予測する。これに使われるのは分岐履歴テーブル（Branch History Table＝BHT）と分岐ターゲットバッファ（Branch Target Buffer＝BTB）である。BHTもBTBも分岐命令のアドレスをインデックスとするキャッシュである。（キャッシュにヒットする場合）BHTの出力は分岐命令が分岐するか否かの予測情報であり，BTBの出力は予測した分岐先のアドレスである。

また，分岐予測の効果を増大させるため投機実行（Speculative Execution）を行うMPUもある。投機実行とは，分岐予測の成功/失敗がわかる以前でも命令を実行させる機能である。しかし，MPUは投機的に実行されている分岐命令の分岐/不分岐が確定するまで（つまり分岐命令のコンプリートまで）リタイアできない。もし分岐予測が失敗すれば，分岐命令以降に実行された命令を放棄して，分岐元から命令の処理をやり直さなければならないからである。投機実行中の命令の結果は，通常リオーダーバッファに格納される。分岐予測が失敗したらリオーダーバッファの該当エントリを無効化すればよい。

ところで，投機実行に限らず，一般的には演算結果はリオーダーバッファに格納されてリタイアを待つが，MIPSのR10000などは（レジスタにネーム後の）物理レジスタを直接更新する。これは，アーキテクチャ的に論理レジスタにも実体があり，物理レジスタは値を一時的な演算結果を保持するものとして区別しているためである。この場合，リオーダーバッファの中には演算結果の格納領域は不要である。R10000では命令のリタイア時に，その命令に割り付けられている物理レジスタの値が論理レジスタに転送される。

歴史的にはR10000方式のほうが古く，かつ一般的のように思える。x86系のMPUのように物理レジスタを実質的なレジスタの本数を増加させる目的でレジスタリネームに使用すると，それを一時的な結果の保存場所に利用することはできない。リオーダーバッファ内に一時的な結果を持つのは姑息な方法のようにも思える。まあ，x86系MPUのシェアは膨大なので，そちらの方式が大勢といわれれば確かにそうであるが。

投機実行を行う場合，分岐条件未確定中のロード/ストアがどのように処理されるかは興味深い。たとえば，ストアを行ったあとで分岐予測の失敗が判明したときキャッシュやメモリに不正なデータが書かれることはないのか不安になる。ロード/ストアがキャッシュ領域に対して行われるものならば，ROBと同等な一時バッファを設けて，ロード/ストア命令のリタイアまで保持すればよい。PentiumではこのバッファをMOB（Memory Order Buffer）と呼んでいる。さて，ロード/ストアが非キャッシュ領域に対して行われるときは事情が異なる。最近のMPUは専用のI/O命令を持っていないため，メモリ空間にI/Oアドレスを割り付けて，そこを非キャッシュで参照することでI/O機能を実現する（メモリマップトI/O）。I/O装置にはリードを行うと内部状態が変化するものもあり，実際には実行されない（分岐予測が失敗する場合の）投機実行中のロードを行うと周辺が誤動作してしまう。つまり，このような場合，投機実行中の非キャッシュ領域へのロード/ストアは実行してはいけない。また，同様に，非キャッシュ領域へのロード/ストアの順序は変更してはいけない。最近のMPUは，ノンブロッキングキャッシュ機能を実装し，ロード/ストアもアウトオブオーダーに行われるが，これはキャッシュ領域に対する場合のみである。

●実質性能（MIPS値）か動作周波数（G/MHz）か

現在のMPU設計の基本方針は，スーパーパイプライン（ステージ数を増加させる）で高い動作周波数を実現し，ステージ数増加による分岐命令の性能低下を分岐予測機能を高度化することによって補う傾向がある。しかし，パイプラインのステージ数の増加によるIPCの低下率が，動作周波数の向上率よりも大きい場合が多々ある。この場合，実質的な性能は低下することになる。しかし，MIPS値やSPECmarkで表された性能よりも，動作周波数の値のほうがユーザーに与えるインパクトが大きいので，実質性能が下がると知りながら，あえて高い動作周波数のMPUを開発することもある。

Pentium4の発表でこういった状況を考えるのにいい機会が与えられたので，少し言及しよう。インテルの正式発表ではないものの，Webサイトに出回っている情報によると，Pentium4の整数性能は1.5GHz動作時に501SPECint2000という。これに対し，Pentium IIIの1GHz動作時の整数性能は428SPECint2000となっている。ベンチマーク全体が1次キャッシュに収まるDhrystoneベンチマークとは異なり，SPECmarkではキャッシュミスによる外部バスサイクルが発生するため，性能がバスの速度に律速され，動作周波数に比例するとは限らないのだが，無理やり，Pentium4の1GHz動作時の整数性能を計算すると334SPECint2000となる。つまり，同一動作周波数ではPentium4の性能はPentium IIIに劣るということ，つまり，IPCが低下しているということになる。あるいは，Pentium IIIの1GHz品はPentium4の1.28GHz品と同性能である。実際にはPentium IIIは1.13GHz，Pentium4は1.4GHzということなので，これではわずかながらPentium IIIの性能のほうが勝る。Pentium4は1.5GHzで動作させることによって，かろうじて優位性を保てるのである。これは，プリデコードした命令をトレースキャッシュに格納する，クロックの両エッジでALUを駆動する，巨大な分岐予測テーブルを備える，といったマイクロアーキテクチャ（NetBurstという）の改良にもかかわらず，パイプラインのステージ数がPentium IIIの倍近い20

ステージになることに伴うIPC低下を補うことができなかったということであろう。また，Pentium4のチップサイズは217mm²で，Pentium Ⅲの2倍以上である。これは，製造原価がPentium Ⅲの2倍以上かかることを意味する。さすがに，Pentium4がPentium Ⅲの2倍の価格で販売されることはないと思うが，コストパフォーマンスは思った以上に悪いようである。2GHz以上の動作周波数を達成したときに初めてPentium4の真価が発揮されるのではないだろうか。

●スーパースカラの実際

典型的なMPUについてスーパースカラの実装方式を見ていこう。ここではインオーダー方式の代表例として日立のSH4とインテルのP5を，アウトオブオーダー方式の代表例としてインテルのP6とMIPSのR10000を紹介する。といったものの，SH4のパイプラインに関しては資料がほとんど存在しないし，x86のことはあまり知らない。ただし，x86に関しては資料は豊富である。SH4は想像，x86は既存の資料の受け売りが多いことは否定しない。多少の誤りは容赦願いたい。

[1]SH4のパイプライン

SH4は，整数ユニット，浮動小数点ユニット，ロード/ストアユニット，分岐ユニットという，4つの基本演算ユニットを備える。この4つのユニットに対し，2命令を同時発行するインオーダーな2ウェイスーパースカラである。なお，ロード/ストアユニットは単純な整数ユニットの役割を持ち，簡単なMOVやNOPなどの命令を0レイテンシで実行できる。このほかに，複雑な命令を実行するためのユニット（というか上述のユニットを組み合わせて利用？）があるようである。ユーザーズマニュアルによると，SH4の命令は利用する内部機能ブロックにより，次の6グループに分類できる。略語の意味は筆者が適当に考えたので間違っているかもしれない。

1）MT（Manipulate T）グループ
2）EX（Integer Execution）グループ
3）BR（Branch）グループ
4）LS（Load/Store）グループ
5）FE（Floating Point Execution）グループ
6）CO（Complex）グループ

これらのグループは上述の演算ユニットに対応しており，同時に実行できる組み合わせは図7のようになっている。なお，各パイプラインは，通常，次の5ステージで処理される。

1）命令フェッチ（I）
2）デコード・レジスタリード（D）
3）実行（EX，SX，F0，F1，F2，F3）
4）データアクセス（MA，NA）
5）ライトバック（S，FS）

これ自体はSH3のパイプラインと同じである

図7　命令の並列実行性

		第2命令					
		MT	EX	BR	LS	FE	CO
第1命令	MT	○	○	○	○	○	×
	EX	○	×	○	○	○	×
	BR	○	○	×	○	○	×
	LS	○	○	○	×	○	×
	FE	○	○	○	○	×	×
	CO	×	×	×	×	×	×

図8　SH4のパイプラインの命令処理のパターン

（1）一般パイプライン

I	D	EX	NA	S
・命令フェッチ	・命令デコード ・発行 ・レジスタリード ・PC相対分岐の分岐先アドレス計算	・演算	・非メモリデータアクセス	・ライトバック

（2）ロード／ストアパイプライン

I	D	EX	MA	S
・命令フェッチ	・命令デコード ・発行 ・レジスタリード	・アドレス計算	・メモリデータアクセス	・ライトバック

（3）特殊パイプライン

I	D	SX	NA	S
・命令フェッチ	・命令デコード ・発行 ・レジスタリード	・演算	・非メモリデータアクセス	・ライトバック

（4）特殊ロード／ストアパイプライン

I	D	SX	MA	S
・命令フェッチ	・命令デコード ・発行 ・レジスタリード	・アドレス計算	・メモリデータアクセス	・ライトバック

（5）浮動小数点パイプライン

I	D	F1	F2	FS
・命令フェッチ	・命令デコード ・発行 ・レジスタリード	・計算1	・計算2	・計算3 ・ライトバック

（6）浮動小数点拡張パイプライン

I	D	F0	F1	F2	FS
・命令フェッチ	・命令デコード ・発行 ・レジスタリード	・計算0	・計算1	・計算2	・計算3 ・ライトバック

（7）FDIV/FSQRTパイプライン

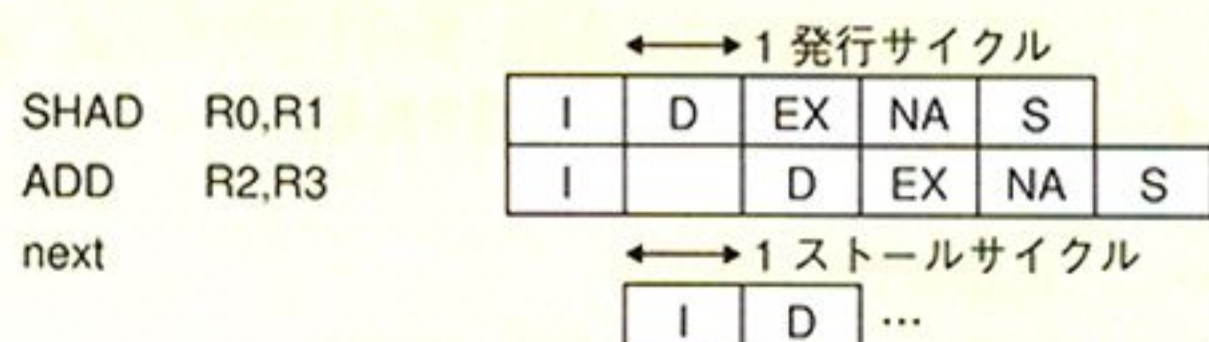
F3
計算：数サイクルかかる

図9　並列実行の状況

(a) 直列実行：並列実行不可命令

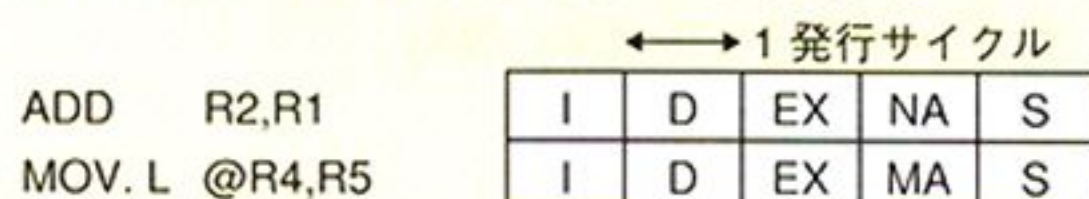
←→1発行サイクル

SHAD R0,R1	I	D	EX	NA	S	
ADD R2,R3	I		D	EX	NA	S
next		I	D	…		

←→1ストールサイクル

EXグループのSHADと同じEXグループのADDは並列実行できない。したがって，先行するSHADのみ発行され，2番目のADDは次の命令と再度組み合わされる。

(b) 並列実行：並列実行可能かつ依存関係なし

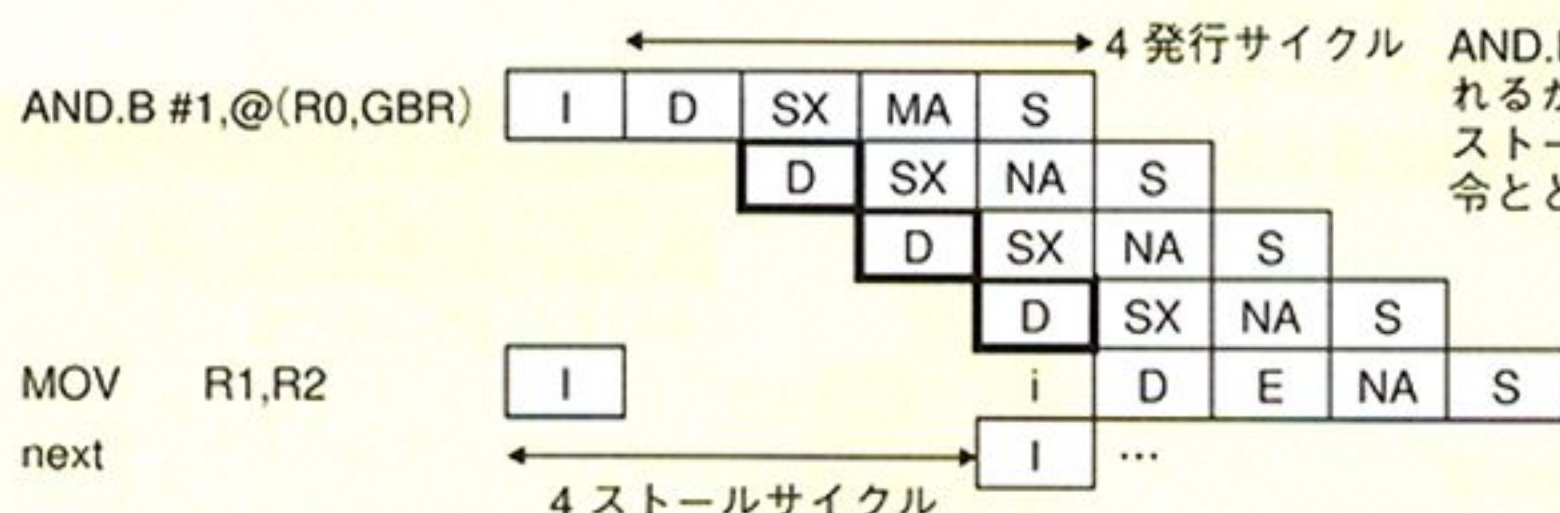
←→1発行サイクル

ADD R2,R1	I	D	EX	NA	S
MOV. L @R4,R5	I	D	EX	MA	S

EXグループのADDとLSグループのMOV. Lは並列実行できる。このとき，2命令のいくつかのステージはそれぞれオーバーラップ可能となる。

(c) 発行レート：マルチステップ命令

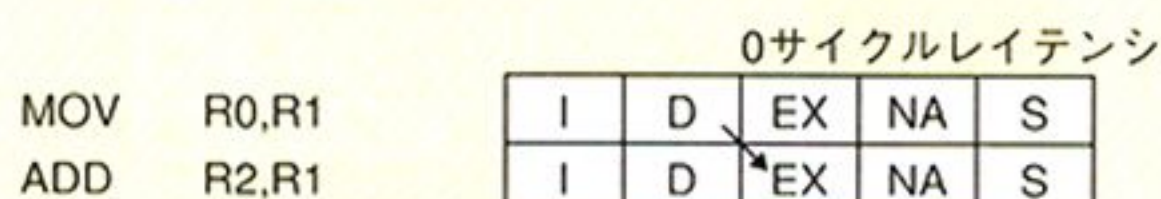
←→4発行サイクル

AND.B #1,@(R0,GBR)	I	D	SX	MA	S				
			D	SX	NA	S			
				D	SX	NA	S		
					D	SX	NA	S	
MOV R1,R2	I				i	D	E	NA	S
next					I	…			

←→4ストールサイクル

AND.BおよびMOVは同時にフェッチされるが，リソースロックのためMOVはストールされる。解放後MOVは次の命令とともに再フェッチされる。

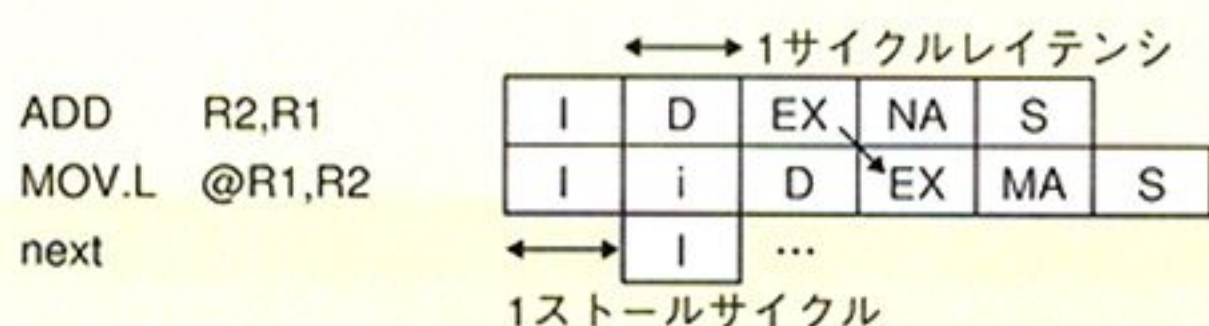
(d) フロー依存関係

0サイクルレイテンシ

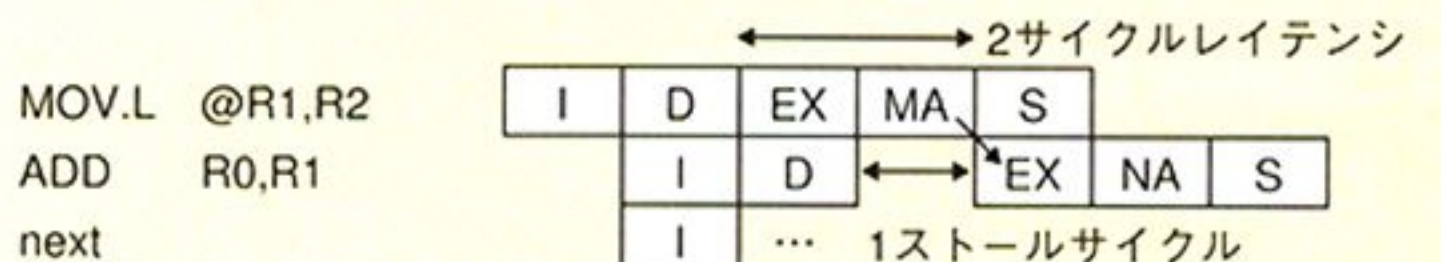
MOV R0,R1	I	D	EX	NA	S
ADD R2,R1	I	D	EX	NA	S

フロー依存関係が存在する場合でも後続命令ADDは0サイクルレイテンシの命令のあとの実行時にはストールされない。

←→1サイクルレイテンシ

ADD R2,R1	I	D	EX	NA	S	
MOV.L @R1,R2	I	i	D	EX	MA	S
next		I	…			

←→1ストールサイクル

MOV.LはADDの結果をそのロードアドレスとして参照するのでADDとMOV.Lは並列実行されない。

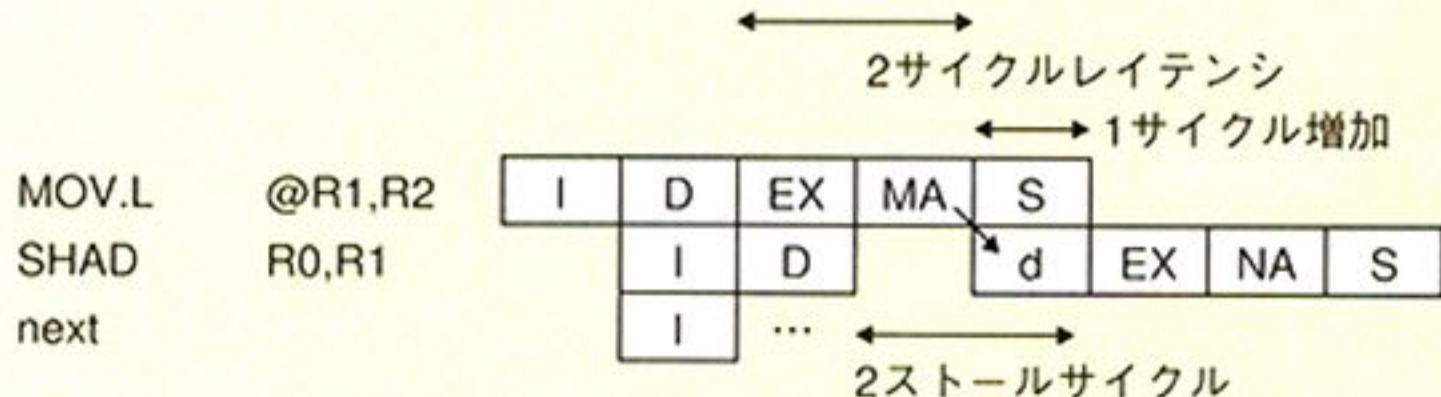
←→2サイクルレイテンシ

MOV.L @R1,R2	I	D	EX	MA	S	
ADD R0,R1		I	D	←→	EX	NA	S
next		I	…	1ストールサイクル		

MOV.Lのレイテンシが2サイクルであるにもかかわらず，ADDは1サイクルの間だけストールされる。この例では，MOV.LとADDは同時にフェッチされないとしている。同時にフェッチされても同様。

←→2サイクルレイテンシ

←→1サイクル増加

MOV.L @R1,R2	I	D	EX	MA	S			
SHAD R0,R1		I	D		d	EX	NA	S
next		I	…					

←→2ストールサイクル

ロードとSHAD/SHLDのシフト量の間のフロー依存関係によりロードのレイテンシは3サイクルに増加する。

(浮動小数点演算が追加されているが)。図8に基本的なパイプラインの命令処理パターンを示す。

ユーザーズマニュアルをちらっと眺めたところでは，2命令を同時発行できる場合は，0レイテンシと1レイテンシの命令の組み合わせだけのようである。片方が0レイテンシならば無条件に，1レイテンシ同士なら，命令のグループが異なり，かつ，レジスタの依存性がない場合に限り，同時発行できる。2命令間が同時発行できないグループにあるとき，実行に必要なハードウェア資源が競合するとき，レジスタの依存関係があるときは同時発行できない。この場合，第2命令は，その後続命令とともに再度組み合わされて，同時発行ができるか否かを決定する。この関係を図9に示す。

マニュアルを読み間違えていなければ，マルチステップ命令(この種の命令はハードウェア資源を独り占めする)が出現したり，レジスタの依存関係がある場合は，SH4のパイプライン処理の効率はシングルパイプラインと大差ないように思える。少しの並列実行を行うために，ハードウェア資源を無駄遣いしているように感じるのは筆者だけであろうか。せめて，整数演算器がもう1個あればもう少し性能は向上するであろう。

レジスタの依存関係がある場合，プログラム上は，同時発行された2命令の第2命令がストールして待たされるように見える。つまり，第1命令のレイテンシだけストールする。レイテンシとは，早い話が実行ステージのサイクル数である。このストール期間は通常1サイクルであるが，厳密には次のように規定されている。

1) フロー依存関係(RAWハザード)が存在するときは，先行する命令の(レイテンシ)サイクル。
2) 出力依存関係(WAWハザード)が存在するときは，先行する命令の(レイテンシー1)または(レイテンシー2)サイクル。
   (a) 単/倍精度FDIV，FSQRTが先行するときは(レイテンシー1)サイクル。
   (b) (a)以外のFEグループの命令が先行するときは(レイテンシー2)サイクル。
3) 次のような逆フロー依存関係(WARハザード)が存在するときは，5サイクルまたは2サイクル。
   (a) FTRVが先行するとき5サイクル。
   (b) 倍精度FADD，FSUB，FMULが先行するとき2サイクル。

ところで，以下にSH4のパイプラインで特徴的な2つの例を示す。

1) 0レイテンシ命令はレジスタの依存関係があってもストールしない。

これはソースフォワード機能と呼ばれる(通常のフォワードとは異なる)。おそらく，デコード時に他方の命令のソースオペランドを読み替えるのであろう。たとえば，

```
MOV  R1,R0   // R0 <- R1
ADD  R0,R2   // R2 <- R0+R2
```

という命令列がある場合，ADDのソースであるR0はR1に変換されて，依存性をなくし，この2命令を1サイクルで実行する。

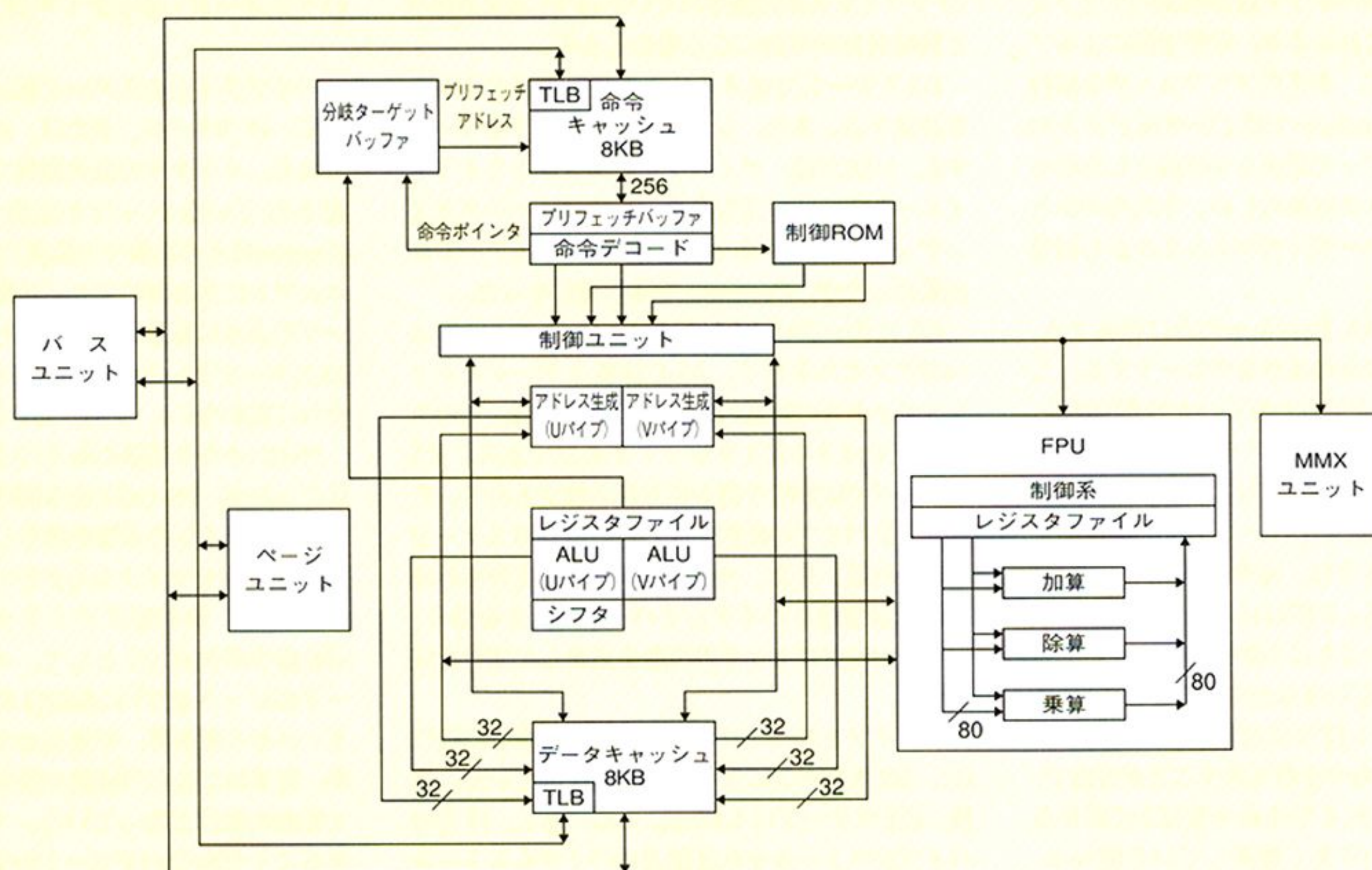
2) 条件分岐命令でオフセットが0の場合は，分岐成立時でも1サイクルで実行できる。

分岐命令が分岐する場合，分岐先のフェッチまでに1サイクルストールするので，レイテンシは2であるが，分岐のオフセットが0(2命令後への分岐)の場合は，分岐先フェッチのオーバーヘッドがなく，1サイクルで実行できる。つまり，スキップ命令として利用できる。たとえば，次のような命令列が考えられる。最近のMPUでも，2命令後への分岐は特別扱いして高速化が試みられているが，これはその先駆けかもしれない(でも，そのような場合は条件MOVE命令を使用することのほうが多いような気もするが)。

```
CMP/GT R1,R2    // if R2>R1 then set T-bit
BT     label  // branch relative if T-bit set
ADD    #4,R3    // if T-bit set then R3<-R3+4
label:
```

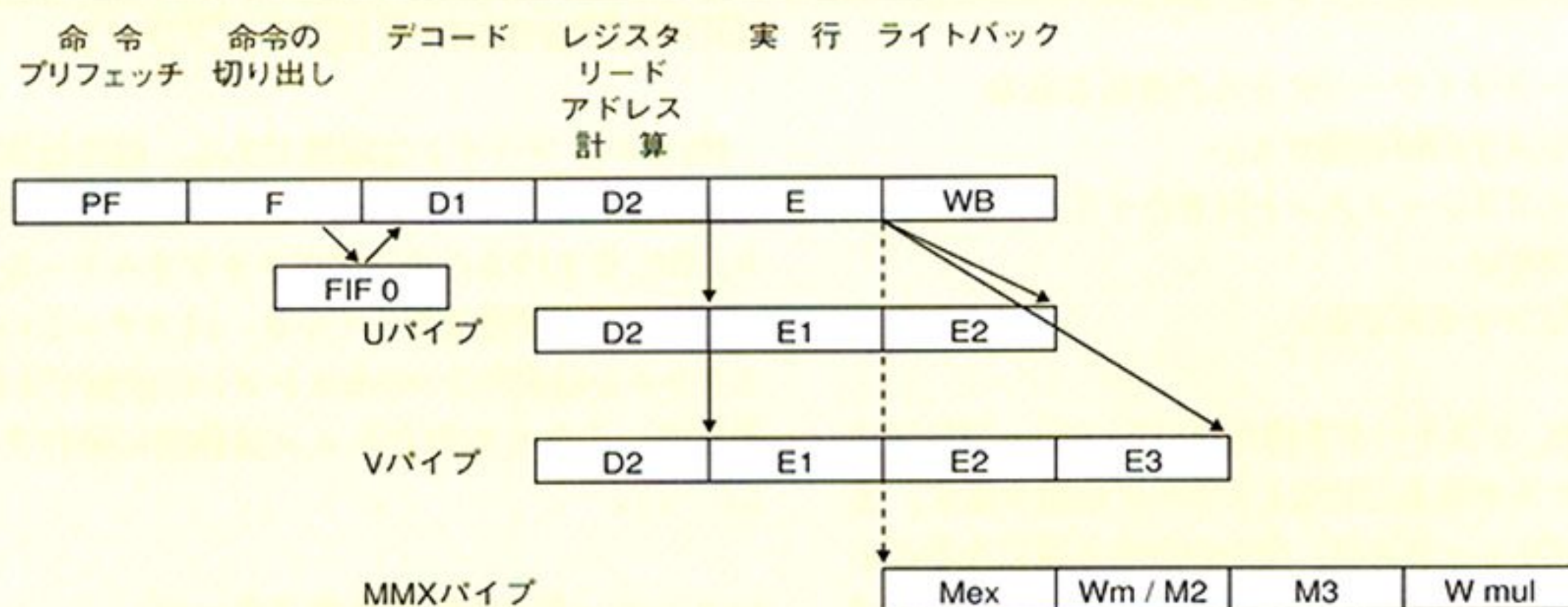
SH4のパイプラインは2命令同時実行に制限が多い。ここら辺りはコンパイラの頑張り次第といえるかもしれない。日立の発表では，SH4の性能は200MHz動作で360MIPSということになっている。これはDhrystone1.1による値であり，他社が採用しているDhrystone2.1では300MIPS程度という噂もある(公式発表はない)。200MHz

図10　P5（Pentium）のブロック図

図11　P5（Pentium）のパイプライン

で300MIPSも達成できれば，インオーダースーパースカラとしては，まずまずの性能であろう。しかし，300MIPSというのも少し眉唾な気がしないでもないが（このパイプライン構造のわりに性能がよすぎる）。

SH4の後継機種であるSH5では，スーパースカラ構造では制御が複雑で動作周波数を上げることができないという理由で，7ステージのシングルパイプラインが採用された。これで，動作周波数はSH4の200MHzから400MHzへと向上するという。しかし，筆者にはSH4のパイプラインが周波数の向上の妨げになるほど複雑だとは思えない（でも規模は大きそう）。むしろ，SH4ではシングルパイプライン並みの効率のスーパースカラでしかなかったので，効率的なシングルパイプラインで作り直したと考えるほうが納得できる。実際，SH5の性能は400MHz動作時に714MIPS（Dhrystone1.1）と発表されており，IPCで見るとSH4もSH5もほとんど同性能と思われる。これが本当ならSH5はとても素晴らしいCPUといえる。個人的にはシングルパイプラインでMIPS/MHzの値が1.785というのは不可能だと思うのだが（どう考えても，インオーダースーパースカラの値だよなあ）。

[2]P5（Pentium）のパイプライン

P5のパイプラインはi486と同様の5ステージから構成される。MMX Pentiumではフェッチステージが1段追加されて6ステージになる。イメージ的にはそのパイプラインが2本並列に動作するインオーダースーパースカラ方式である。2つの汎用整数パイプラインに加えて，パイプライン化されたFPU演算を同時に実行できる。これら2つの整数パイプラインはUパイプとVパイプと呼ばれる。Uパイプではすべての命令を実行できる。一方，Vパイプでは単純な命令のみを実行できる。同時発行可能な2命令をデコードしたとき，（プログラムの順番で）先行する命令はUパイプで，後続する命令はVパイプで実行される。イメージ的には，Uパイプが常に動作していて，後続命令が同時実行可能な場合のみVパイプも使用する，といったところか。図10にP5のブロック図を示す。なお，5つのパイプラインステージの内訳は次のようになっている。また，図11にパイプラインステージを図示する。

PF　プリフェッチ
F　フェッチ（MMX Pentiumのみ）
D1　命令デコード
D2　アドレス生成
EX　実行…ALU演算とキャッシュアクセス
WB　ライトバック

PFステージでは命令キャッシュまたはメモリから命令がプリフェッチ（先取り）される。Pentiumでは，従来のi486などとは異なり，キャッシュが命令キャッシュとデータキャッシュに分かれているので，プリフェッチがデータ参照と競合しない。PFステージでは，2つの独立なラインサイズ（16バイト×2）の組み合わせのプリフェッチバッファが分岐ターゲットバッファ（BTB）と結合されて動作する。条件分岐命令に行き当たるまではプリフェッチは逐次的に進む。条件分岐命令が

プリフェッチされるとBTBで分岐予測が行われ，片方のプリフェッチバッファは分岐先のプリフェッチに使われる。これにより，分岐予測によるプリフェッチと同時に，本来のプリフェッチを続行できる。MMX Pentiumでは4つの16バイトのプリフェッチバッファで最大4つの独立した命令の流れをプリフェッチ可能らしい。本当なのかと疑ってしまうが，ユーザーズマニュアルからの受け売りである。

Fステージは MMX Pentiumのみに存在する。このステージでは命令の長さをデコードする。これは従来D1ステージで行われていた処理である。プリフィクスのデコードもFステージで行われる。

MMX Pentiumでは，さらに，FステージとD1ステージの間に命令キュー（FIFO）が存在する。FIFOが空のときは，命令は遅延なしでD1ステージに渡される。FIFOは4命令分用意されていて，各サイクルごとに2命令を格納可能である。FIFOからは2組の命令が引き出されてD1ステージに渡される。FIFOは通常命令で満たされているので，常に2命令を取り出すことが可能で，1サイクルで実行される平均命令数は2に限りなく近づく。FIFOがうまく機能している限りは，命令フェッチとFIFOからの命令の切り出しでストールは生じない。

D1ステージでは連続する2命令を同時にデコードし発行する。同時に発行できる命令の組は次のような関係にあるものである。

a）ハードワイヤード化された単純な命令
b）レジスタの依存性がない
c）ディスプレースメント付きとイミディエートの組でない
d）プリフィクスでない

なお，Fステージを持たないPentiumではプリフィクスがあるだけD1ステージを繰り返す。また，プリフィクスは，ほかの命令と組になることはなく，Uパイプのみで実行される。すべてのプリフィクスが発行されると，ベースとなる命令（プリフィクスが付加されていた命令）は次の命令と同時発行が可能になる場合もある。

D2ステージではメモリオペランドのアドレスを計算する。また，レジスタオペランドをリードする。i486では，ディスプレースメントとイミディエートを同時に含む命令，または，ベースとインデックスを持つ命令はもう1回D2ステージが必要だったが，Pentiumでは不要になった。

EXステージはALU演算とデータキャッシュへのアクセスを行う。ALU演算とデータキャッシュアクセスの両方の処理が必要な場合，このステージではさらに1クロックが必要である。EXステージでは分岐予測の正当性の検証も行う。ただし，Vパイプの条件分岐の検証はWBステージで行われる。また，マイクロコードで実行される複雑な命令はUパイプとVパイプの両方を使う。

WBではプロセッサの状態を更新して実行を完了する。

UパイプとVパイプで実行される命令は同時にD1，D2ステージに入り，同時に抜けていく。当然，EXステージにも同時に入る。もし，片方のパイプがストールすれば他方のパイプもストールする。両方のパイプの命令がWBステージに達するまで，新たな命令はEXステージに入ってこられない。こうして，インオーダー完了を実現している。

[3]P6（Pentium II）のパイプライン

P6のパイプラインは複雑である。動作周波数を上げるためにスーパーパイプライン構造を採り，IPCを上げるためにアウトオブオーダーなスーパースカラ構造を採っている。14ステージのパイプラインは次の3つのセクションに分割できる。そして，これらのセクションは独立に動作する（図12）。

a）インオーダーな前処理（8ステージ）
b）アウトオブオーダー実行（3ステージ）
c）インオーダーなリタイア（3ステージ）

パイプラインのステージ数は，機能分割のやり方で，12ステージ，または，10ステージという説もある。インテルの公式資料ではステージ数は明記されていなかったと記憶している（最近のPentium4との比較での発表では10ステージ）。マニュアルにある図のステージ数を数えると13ステージのようにも思えるが，実行ステージを抜いて12ステージという解釈が有力である（でも本稿はそれに従わない）。まあ，たいした問題ではない。

P5は（かなり制限のある）2命令同時発行のMPUだったが，P6では3命令同時発行になった。単にパイプラインの本数を増やしただけでなく，P6はx86命令をマイクロOPというRISC風の命令に変換し，効率的にパイプラインを処理する。x86命令の欠点（?）として，エンコード（命令コードのビット並び）に規則性がないこと，レジスターメモリ間演算，可変長命令などが挙げられるが，従来はこれらの特徴が効率的なスーパースカラ処理の妨げとなっていた。マイクロOPを導入することで命令のデコードが容易になり，RISC並みのパイプライン効率を得ることができる。

図13にP6の機能ブロックを示す。この図を基に命令処理の過程を説明する。

1）x86命令の変換

x86命令からマイクロOPへの変換はパイプラインの最初の8ステージで行われる。まず，分岐ターゲットバッファ（BTB）が指し示す位置の64バイト（キャッシュ2ライン分）のコードを命令キャッシュから読み込む。その中で，最初にあるx86命令の先頭から16バイトのコードを取り出して並列動作する3つのデコーダに渡す。x86アーキテクチャは可変長命令を採用し，プリフィクスを付加することで（理論上）無限長の命令を生成することができる。P6は1命令の長さを平均5バイトと仮定しているのであろう。もっとも，16バイトのコードのうち，この時点で命令の切れ目は不明なので，3つの命令デコーダは16バイトすべてを受け取ると推測される。Pentium系のMPUでは，命令が16バイト境界にまたがる場合は命令の実行効率が落ちるといわれているが，それはここに原因があると思われる。

さて，P6の命令キャッシュは1ラインが32バイトなので，その中の任意の位置から始まる16バイトのコードを得るために，2つのラインが同時に読み込まれる。そして，これら3種の命令デコーダがx86命令をRISC命令によく似たマイクロOPに変換する。3つのデコーダのうち，2つが単純デコーダ，残りが複雑デコーダである。単純デコーダはひとつのx86命令をひとつのマイクロOPに変換する。複雑デコーダはひとつのx86命令を1から4個のマイクロOPに変換する。特に複雑な命令は複雑デコーダでもデコードできず，そこを通過してマイクロコード命令シーケンサ（MIS）に渡される。MISは必要なだけのマイク

図12　P6（Pentium II）のパイプライン

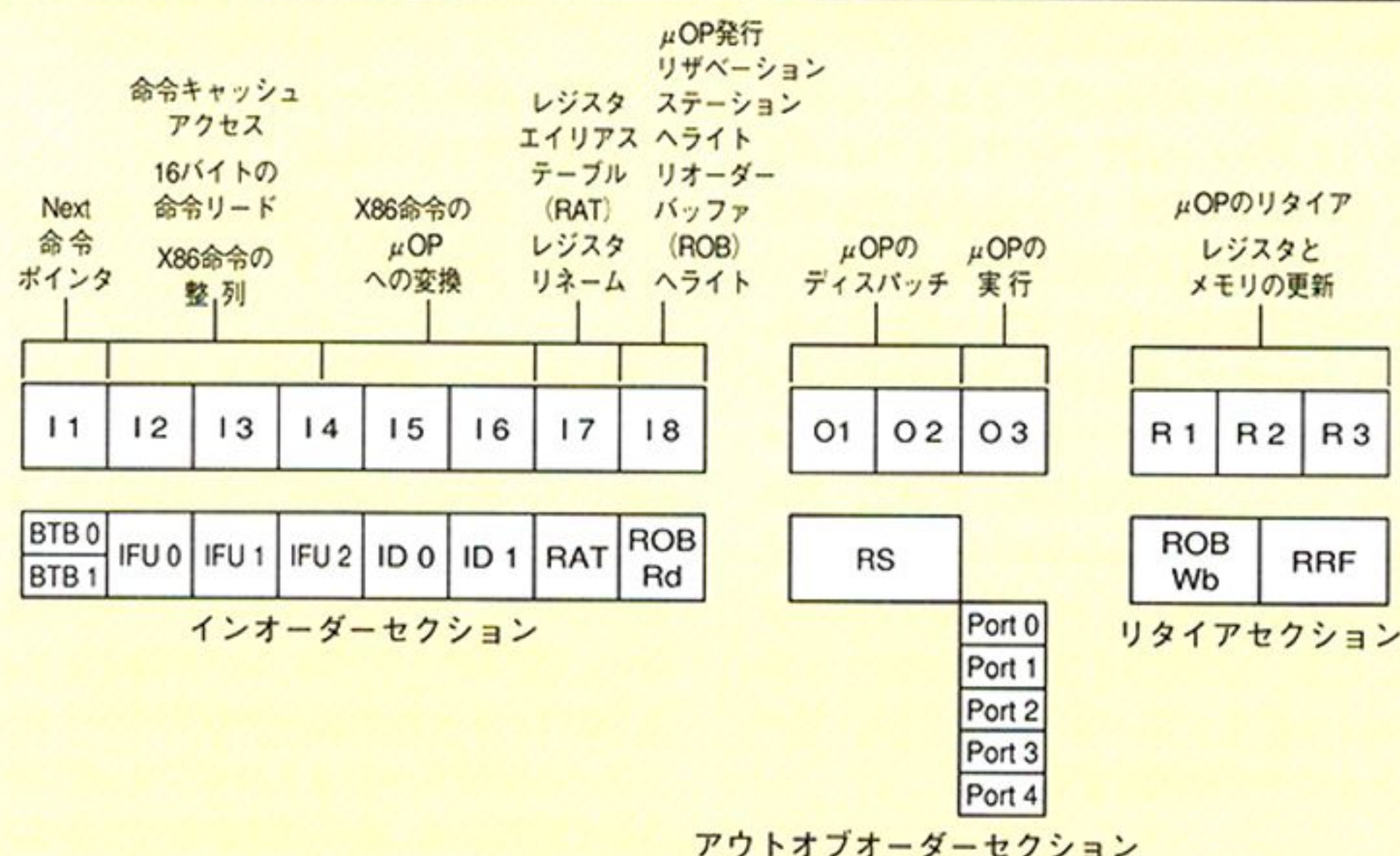
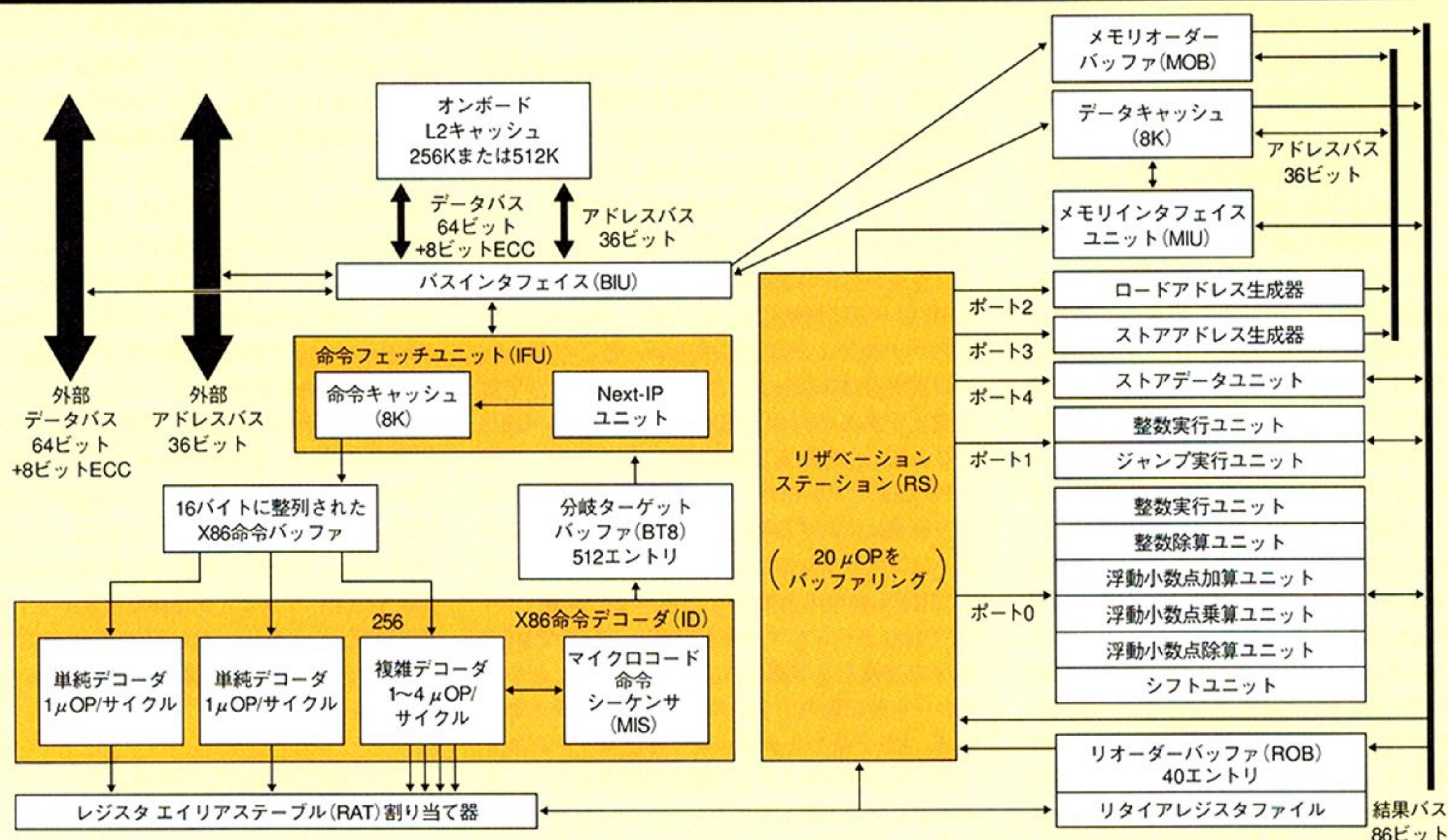



図13 P6 (Pentium II)のブロック図

ロOPを生成する。

たまたま複雑な命令が単純デコーダに割り当てられた場合は，そこから複雑デコーダまたはMISに渡される。ここでのデコードの遅れはリザベーションステーションで吸収されるので，命令の実行には影響ない。

単純な命令と複雑な命令のデコーダへの割り当てが完璧な場合は，1サイクルごとに6つのマイクロOPを生成する。しかし，平均的には1サイクルごとに3つのマイクロOPが生成される。これを根拠にインテルはP6を3ウェイスーパースカラと呼んでいる。

2)レジスタリネーム

マイクロOPに変換されたx86命令は，パイプラインの第7ステージでレジスタエイリアステーブル(Register Alias Table：レジスタ読み替え表＝RAT)に送られてレジスタリネームが行われる。ここで偽の依存性(WAWハザードなど)を解消する。x86アーキテクチャは汎用レジスタ(論理レジスタ)が8本しかないので，レジスタの依存関係は生じやすい。それを軽減させるため，P6では40本の物理レジスタを持つ。つまり，P6は内部的に40本の汎用レジスタを持っていることになる。

レジスタリネームでは真の依存性(RAWハザードなど)は解消できない。P6ではレジスタのフォワーディングを行うのでそのペナルティを軽減できる。

3)アウトオブオーダー実行

レジスタリネームが完了すると，プログラムの順序で，マイクロOPはリオーダーバッファ(ROB)に送られると同時にリザベーションステーションにキューイング(待ち行列に入れる)される。これはデコーダと実行ステージの中間に位置する。リザベーションステーションは最大20のマイクロOPを蓄えることができ，11個の実行ユニットに対して，1サイクルで最大5つのマイクロOPを発行できる(入力ポートが5つあるため)。もっとも，典型的なx86の命令列では1サイクルに発行できるマイクロOPはたかだか3命令といわれている。

リザベーションステーションは，ソースオペランドが使用可能になったか，実行ユニットが空いたか，依存性が解消できたかを調べて，用意ができたマイクロOPをアウトオブオーダーに発行する。アウトオブオーダーにコンプリートするマイクロOPの結果は一時的なバッファ(ROBやMOB)に格納され，ROBの状態を参照しながら，プログラムの順序にレジスタやメモリに書き込まれる。

ROBは40エントリからなる254ビット幅のバッファである。254ビットの内訳は，2つのオペランド，実行結果，多くの状態ビットである。ROBには整数と浮動小数点のマイクロOPの両方が格納される。この処理はパイプラインのリタイアのステージで行われる。

4)リタイア

ROBは実行状態と各マイクロOPの結果を保持する。マイクロOPは先行するマイクロOPがすべてコンプリートしたことがわかって初めてリタイアし，結果をレジスタやメモリに書き込む。この動作をコミットともいう。P6では1サイクルに最大3つのマイクロOPをリタイアできる。これはデコーダが1サイクルに発行できる平均的なマイクロOPの個数(3命令)と釣り合いが取れている。

オペランドのフォワーディングのために，それぞれの実行ユニットの結果はすべてリザベーションステーションに戻される。実行ユニットの結果はROBにも戻されて，リタイアの準備ができたか否かを決定する。

レジスタに対する結果はROBに書き込まれるが，メモリに対する結果はメモリオーダーバッファ(MOB)に書き込まれ，対応するマイクロOPがリタイアするまで一時的に格納される。メモリライトを生じるマイクロOPがリタイアして初めてMOBはメモリにデータを書き込む。

5)分岐予測

P6はパイプラインのステージ数が多いので分岐予測は必須である。分岐予測を誤った場合のペナルティは4～15サイクルである。これはかなりの性能低下になるので，高度な分岐予測が要求される。

P5(Pentium)と同様，P6は分岐ターゲットバ

ッファ（BTB）を採用する。予測方式は分岐履歴ビットによる。ひとつの分岐先アドレスに対して，過去4回分の履歴を記録しておき，それに従って予測する。これは基本的にP5と同じで，4回のループならほぼ100％の分岐予測が可能だという。BTBにヒットしない分岐命令はオフセットの正負などから静的な分岐予測を行う。インテルの主張では，分岐予測の正確さは，P5が80％だったのに対して，P6は90％だという。逆にいえば，分岐予測を誤る確率は20％から10％へと半分になったということである（数字のマジックだな）。

P6で採用している分岐予測の方式は2レベル適応履歴アルゴリズムというものであるが，詳細は明らかにされていない。ただし，命令キャッシュのラインごとに4つの分岐先アドレスをBTBで予測することが公表されている。

ちなみに，先に示したNetNewsで報告されたDhrystoneベンチマークの計測結果では，BTBのミス率は33％という。分岐予測自体の正確さは98％というからBTBにヒットする限り，分岐予測はうまく働いているようである。ただし，Dhrystoneのような単純なプログラムでBTBに67％しかヒットしないというのは納得がいかない。BTBは512エントリなので，1回目のループですべての分岐命令はBTB内に取り込まれて，あとは98％の確率で分岐予測が成功するというシナリオを誰でも思い浮かべるはず。おそらく，BTBの処理アルゴリズムに欠陥があるのかもしれない。

6）投機実行

P6も分岐予測を有効に活用するために投機実行を行う。P6では，分岐予測が失敗した場合の回復処理は，投機的に実行された命令に対するROBのエントリを無効化することで実現している。P6では，ほかの多くのMPUと同様に，ひとつ以上の分岐の方向を予測し実行していくという，多重レベルの投機実行を許している。ただし，ROBが一気に無効化されるため，分岐予測失敗時のペナルティは非常に大きい。ところで，P6の投機実行は5命令までという記述をどこかで見た覚えがあるのだが，記憶違いか。あるいはP5のことだったのかもしれない。

P6ではサブルーチンに対するCALL/RETの組を高速に実行する機構を持っている。サブルーチンはプログラムのさまざまな場所から呼ばれるのでRET命令の分岐先を予測するのは難しい。P6ではリターンスタックと呼ばれる機構でRET命令の分岐先を予測する。これはCALL命令のデコード時に戻りアドレスを格納するスタックである。RET命令をデコードするとリターンスタックにあるアドレスから分岐先を取り出して，そのアドレスを予測したアドレスとして命令フェッチする。物理的なスタックの内容はほかの命令で変更されるおそれがあるので，リターンスタックのアドレスはあくまでも予測値でしかないことに注意。なお，これはスタックキャッシュとして昔から知られている手法である。

P6は，Pentium Pro，Pentium II，Pentium III（Coppermineまで）で使用されてきたマイクロアーキテクチャであるが，最新版のPentium4（Willamette）では，マイクロアーキテクチャの変更が行われた。高い動作周波数（1GHz以上）を実現するために，パイプラインは20ステージとさらに深くなった。そのため，分岐予測が失敗した場合のペナルティが増加するが，BTBを4000エントリにして分岐予測の成功率を上げているという。x86命令からマイクロOPへの変換は，命令キャッシュのリフィル時に行い，変換に要するステージ数を節約している。次に説明するR10000もそうであるが，命令のデコードを容易にするために，命令をプリデコードした結果を命令キャッシュに格納するという手法が今後の流行になるかもしれない。そういえば，TransmetaのCrusoeもx86命令をVLIW命令に変換してDRAM上にマッピングされた命令キャッシュ（?）に格納している。AMDはK6ですでにプリデコードしてある命令長情報をキャッシュに入れる構造になっていた。

[4]R10000のパイプライン

図14にR10000の機能ブロック図を示す。図を見てわかるようにR10000は，演算器として2個の整数ALU，アドレス計算（ロード/ストア）ユニット，浮動小数点加算器，浮動小数点乗算器，浮動小数点除算器（平方根器を兼ねる）を持

図14　R10000のブロック図

図15 R10000のパイプライン

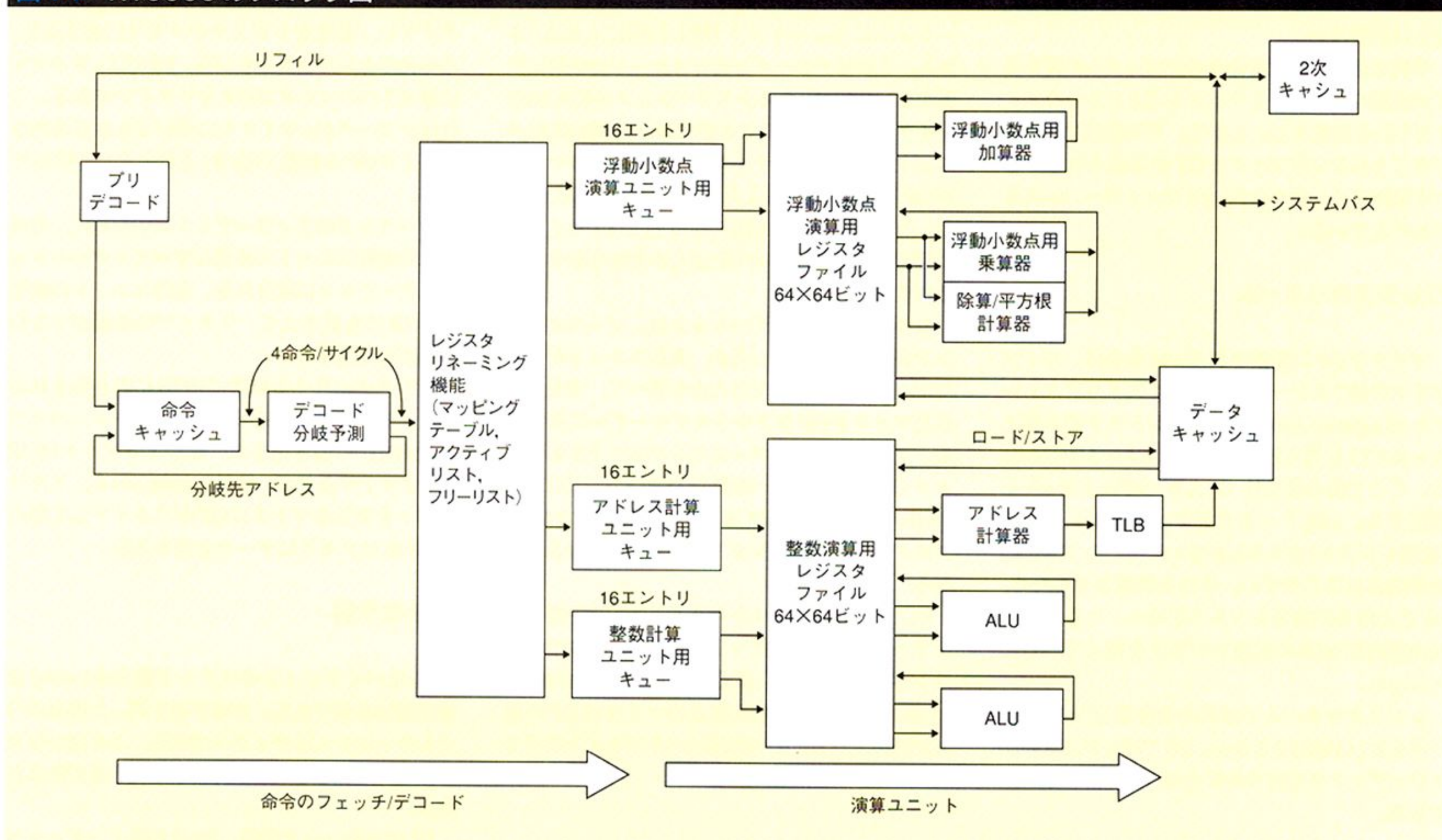
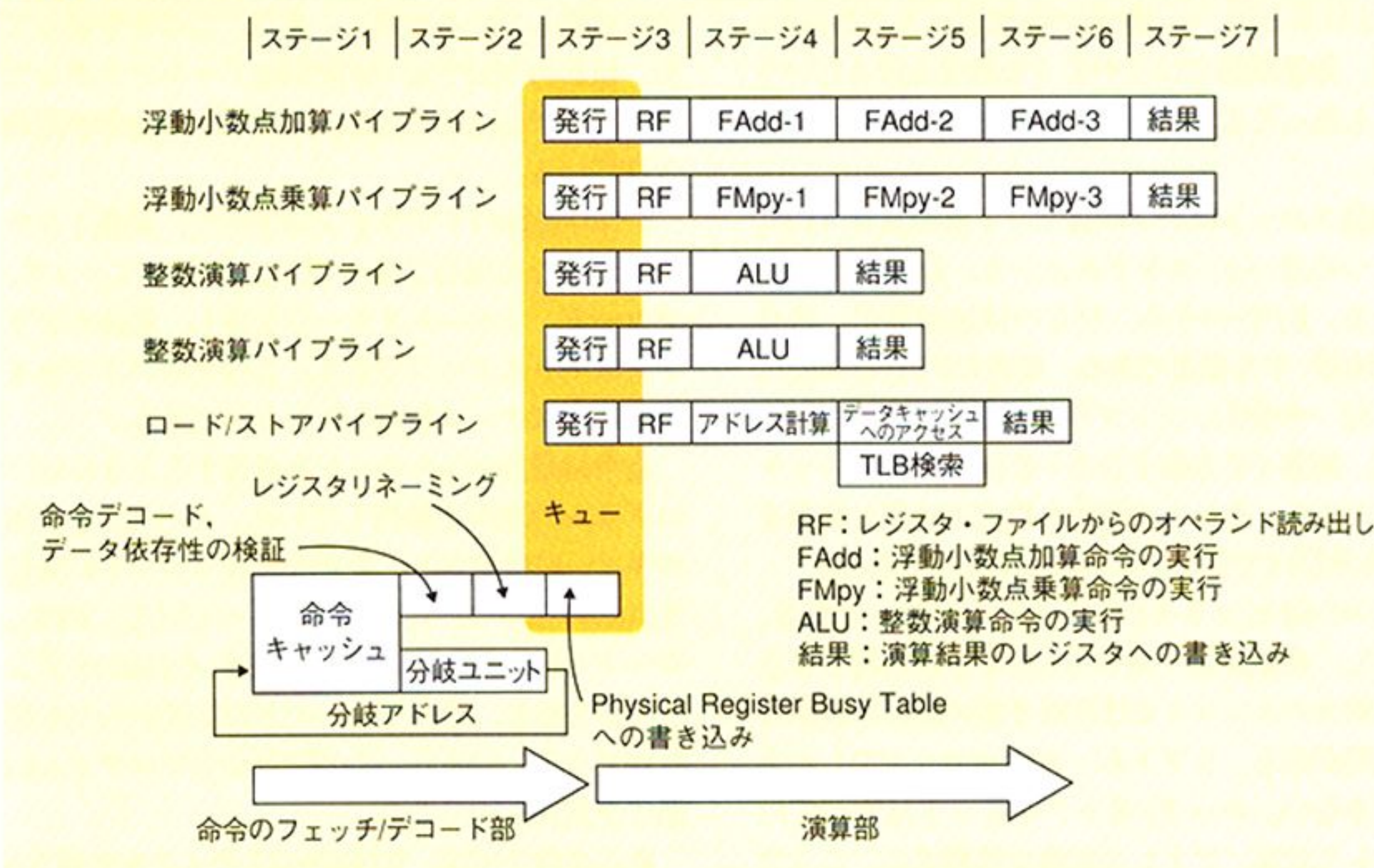
	ステージ1	ステージ2	ステージ3	ステージ4	ステージ5	ステージ6	ステージ7
浮動小数点加算パイプライン			発行 RF	FAdd-1	FAdd-2	FAdd-3	結果
浮動小数点乗算パイプライン			発行 RF	FMpy-1	FMpy-2	FMpy-3	結果
整数演算パイプライン			発行 RF	ALU	結果		
整数演算パイプライン			発行 RF	ALU	結果		
ロード/ストアパイプライン			発行 RF	アドレス計算	データキャッシュへのアクセス / TLB検索	結果	

つ。各演算器には，それぞれ16エントリからなる整数キュー，アドレスキュー，浮動小数点キューが接続されている。命令は各キューから対応する演算器に対してアウトオブオーダーに発行される。R10000は1サイクルで，4命令を命令キャッシュからリードし，最大4命令をデコード/レジスタリネーム可能である。デコードされた命令は32エントリのアクティブリストと呼ばれるリオーダーバッファに格納され，インオーダーなリタイア（R10000ではグラジュエート＝卒業と呼ぶ）を管理する。1サイクルで最大4命令をリタイアできる。演算器は6個備えているが，浮動小数点の乗算器と除算器は入力が共通なので，一度に最大5命令を発行することができる。その意味で5ウェイのスーパースカラであるが，デコード自体は1サイクルで最大4命令なので一般には4ウェイのスーパースカラと呼ばれる。

図15にR10000のパイプラインを示す。一応，7ステージパイプラインの体裁をとっているが，命令のデコード/レジスタリネーム/キューへの格納（ステージ1，ステージ2）までと，発行/実行/結果の格納（ステージ3～ステージ7）までのステージは独立に動作する分離（decoupled）方式である。アウトオブオーダー方式のスーパースカラとしては珍しくない。

R10000のアーキテクチャをMIPS社はANDES（Architecture with Non-Sequential Dynamic Execution Scheduling）と呼び，アウトオブオーダー実行，分岐予測，投機実行などの総称としている。レジスタリネームは32個の論理レジスタを64個の物理レジスタへ割り当てる。割り当て可能な物理レジスタはフリーリストと呼ばれるテーブルで管理される。物理レジスタは整数と浮動小数点の2系統が用意されている。分岐予測は，512エントリ×2ビットの情報で過去2回の分岐/不分岐の履歴を保持し，予測した方向に投機実行する。投機実行は，アクティブリストが一杯になるか，レジスタリネームのためのフリーリストが空になるまで，あるいは分岐命令を4個デコードするまで（つまり4回続けて分岐予測するまで）継続される。分岐予測のたびに，その時点での論理レジスタと物理レジスタの対応表（マップテーブル）のコピーを残しておき，分岐予測が失敗すると，そのコピーに基づいてパイプラインを分岐元の状態に戻す。R10000では演算はすべて物理レジスタに対して行われ，アーキテクチャ上のレジスタ（論理レジスタ）にはリタイア時に対応する物理レジスタから書き戻される。

なお，整数キューとアドレスキューは，基本的には，整数レジスタが対象なので，2種類持つ必然性は少ない。しかし，ノンブロッキングキャッシュの実現を容易にするためにアドレスキューが特設されているものと推測される。もっとも，MIPSの命令セットにはデスティネーションレジスタが浮動小数点レジスタのものもあるので，そのせいかもしれない。

以下にR10000のパイプラインの詳細について説明する。

1)命令フェッチ

各サイクルごとに，R10000は32Kバイトの2ウェイセットアソシアティブ構成の命令キャッシュから4命令をフェッチできる。命令キャッシュに格納されている命令は37ビット長で，命令キャッシュのリフィル時に32ビット長の命令をプリデコードしたものである。この余分な5ビットによって各命令を分類し，その命令が実行されるユニット情報を付加することで命令デコードを効率的に行える。このプリフェッチ/プリデコードステージはパイプラインステージの中には数えられない。余談であるが，ひとつの命令長である32ビット（4バイト）が37ビットに拡張されて格納されるので，命令キャッシュの容量は正確には37Kバイトである（豆知識以上の意味はないなあ）。

2)命令デコード

命令キャッシュからフェッチされた命令は，レイテンシが2ステージの命令デコーダに渡される。実際のデコードは最初のステージで行われ，2番目のステージではレジスタリネームが行われる。

MIPSアーキテクチャではレジスタの本数は整数用と浮動小数点用にそれぞれ32本である。これは論理レジスタと呼ばれる。R10000はさらにレジスタリネーム用に整数用と浮動小数点用の物理レジスタをそれぞれ64本備えている。これらのレジスタファイルが動的にリネームされ，64本の物理レジスタのどれかが32本の論理レジスタを指し示すようにマッピングされる。プログラム的には32本の（論理）レジスタしか見えてないが，MPU内部では2倍の本数の（物理）レジスタが処理結果を保持している。

レジスタリネームはアウトオブオーダーな投機実行を実現するために重要である。R10000は演算の中間結果や投機実行の結果を，この不可視な物理レジスタに保持する。これらの結果はすべての依存性が解決され，投機的な実行経路が消え去ったときにプログラムから見えるようになる。

MPU内部でなにが起きているかを管理するために，R10000はすでに使用されているレジスタ（の番号）を保持するアクティブリストと利用可能なレジスタ（の番号）を保持するフリーリストを用意している。アクティブリスト内のレジスタは2つの状態を取ることができる。ひとつは「アクティブ」である。つまり，実行中の命令で使用されている状態である。もうひとつは「コンプリート」である。つまり，命令実行の最終結果を示す状態である。ある瞬間には最大32命令が「アクティブ」状態にある。「コンプリート」状態になり結果がグラジュエートすると，不要になったレジスタはアクティブリストから削除されフリーリストに返却される。投機実行はフリーリストに利用可能なレジスタがあり，レジスタリネームが可能な限り継続できる（アクティブリストに空きがあることも必要である）。

レジスタリネームは，また，分岐予測において重要な役割を果たす。つまり，分岐が誤って予測されたときに，投機的な実行経路を迅速に破棄する役割がある。R10000は分岐ごとに，最大4つの分岐の深さまで，投機実行の入れ子が可能である。実行経路の分岐点ごとに，レジスタ状態のコピーを持つ。そのコピーは，MIPSの表現によると，その時点で存在しているレジスタリネームマップ（割り当て表）のシャドウマップと呼ばれる。後に分岐予測が失敗したことが判明しても，R10000はバッファのフラッシュとかレジスタのクリアを行う必要はない。単に（予測を誤った分岐命令に対応する）最適なシャドウマップを現状のレジスタリネームマップにコピーし直すだけでよい。そのとき，無効な結果を保持しているレジスタはフリーリストへ返却される。この操作は1サイクルで行われる。このため，誤った分岐予測のペナルティは1～4サイクルになる。これは

R10000が何回誤った予測を行ったかに依存する。最大の入れ子である4個の分岐予測を誤った場合が最悪の4サイクルになる。

分岐予測も動的に行われる。R10000は2ビットの情報で各分岐の履歴を記憶しておく。これは4種類の状態，すなわち，「強く分岐する」，「弱く分岐する」，「弱く分岐しない」，「強く分岐しない」である。MIPSによるとR10000の分岐予測の正確さは90％以上であるという。これは，典型的な動的な命令列において，平均的に6命令に1回分岐が現われるという根拠に基づいているらしい。

3）命令実行

R10000は各サイクルごとに，最大4命令をフェッチし，最大4命令をグラジュエートするが，その中間には5つの実行ユニットがある。このため，可能性としては，各サイクルごとに5命令を同時発行，実行，コンプリートできる。このため，R10000は4ウェイスーパースカラとも5ウェイスーパースカラとも呼ばれる。しかし，命令の処理数に関するこの不整合は偶然ではない。ピークのバンド幅を大きくしておくことで，内部資源の割り当てが効率よく行えるし，将来の拡張の余地を残している。という説明はもっともらしいが，本当にピーク時のバンド幅を考慮するなら，整数演算ALUもFPUも4個ずつ用意すべきであろう。実際，後継機種ではそのような構成を採るという動きもあったようだが，いまだ実現には至っていない。

機能ユニットは2つの64ビット整数演算ALU，ひとつのロード/ストアユニット，2つのFPUからなる。FPUのうち，ひとつは加減算用，残りは乗除算/平方根用である。後者のFPUは実際には，同一の発行/コンプリート論理を共有する，乗算，除算（平方根を含む）を行うサブユニットの組である。それらは浮動小数点の乗算と除算を（同時発行はできないが）並行に実行できる。

2つの64ビットALUはほとんど同一である。ただし，乗除算は一方のALUでしか処理できない。他方のユニットには分岐予測の結果を確かめる論理がある。シフトも一方のユニットでしか実行できない。ロード/ストアユニットはすべてのアドレス計算，アドレス変換を処理する。ここで問題となるのはNOP命令である。MIPSアーキテクチャのNOP命令は「SLL r0,r0,r0」，つまりシフト命令である。プログラム中にかなりの頻度で出現するNOP命令が片方のALUでしか実行できないというのは性能上問題である。これを回避するため，R10000ではNOP命令はプリデコード時に「ADD r0,r0,r0」などの並列実行可能な命令に変換していると聞く。また，これを考慮してか，最新のMIPS32/MIPS64アーキテクチャではSSNOP（SuperScalar NOP）なる命令が定義されている。

5つの実行パイプラインはすべて，最低1ステージからなる実行ステージと，上述のフェッチ，デコード，リネームステージを持ち，最後がグラジュエートステージである。このためパイプラインの最小ステージ数は5ステージである。

命令は最初の3ステージを通過するときにはプログラムの順序を維持している。そして，3種類のキューに格納され，最適な実行ユニットに発行されるのを待つ。これらのキュー（ALU，FPU，ロード/ストアユニット用）のそれぞれは16エントリからなり，そのキューのどの位置からでも発行ができる。つまり，この時点からプログラムの順序を維持しなくなる。

ある条件下では，R10000は1サイクルで最大5命令をキューからアウトオブオーダーに発行できる。しかし，多くの場合は，命令の依存性に応じて1～4命令を発行する。IPCから察するに，平均は2命令前後であろう。

RISCチップでは当たり前のことであるが，ほとんどすべての命令は1サイクルで実行される。浮動小数点の加算，減算，乗算，比較，型変換には2サイクル（レイテンシ）必要である。もっとも，加算と乗算では結果の転送にもう1サイクル必要である。もっと複雑な浮動小数点演算は繰り返し処理で実現され，多くのサイクルを消費する。

ロード処理はデータキャッシュにヒットするときに2サイクルかかる。また，ロードは投機的にアウトオブオーダーで実行される。これに加え，ノンブロッキングキャッシュ構造によりロードを効率的に処理する。ノンブロッキングとは，ロードがキャッシュにミスしてもストールすることなく先に進める技術である。アウトオブオーダーで命令の追い越しが可能な場合，特に効果的である。R10000では最大4つのロードをノンブロッキングで実行できる。

図16-1　サンプルプログラム

	アセンブラのコード		実際の演算	リネーム	使用ユニット	サークル	備考
1	DSLL	r3, r2, 2	p9←p2shift left2	r3=p9	ALU1	1A	
2	LW	r4, 0x8（r3）	p10←mem（p9+8）	r4=p10	L/ST	1B	命令1の r3と真の依存性
3	ADDI	r5,r2,0x34	p11←p2+'34'	r5=p11	ALU	1C	
4	SUB	r3,r1,r5	p12←p1-p11	r3=p12	ALU	1D	命令1のr3と出力依存（偽） さらに，命令2の r3と逆依存
5	XORI	r2,r1,0xFF	p13←p1 xor 'FF'	r2=p13	ALU	2A	
6	BEQ	r4,r2,label	branch if p10=p13	none	ALU1	2B	条件分岐（TAKENと予測）
7	NOP	-	no operation	none	ALU	2C	遅延スロット
8	MULT	r3,r4					分岐がTAKENするならデコードされない
9	MFLO	r2					
…							
label:							
10	AND	r3,r1,r2	p14←p1 and p13	r3=p14	ALU	3A	分岐ターゲットキャッシュにヒットすると仮定
11	LB	r4,0（r3）	p15←mem（p14+0）	r4=p15	L/ST	3B	

命令	1	2	3	4	5	6	7	8	9	10	
1 DSLL	dec	issue	ALU 1	write							先行する命令がすべてグラジュエートしているならば，コンプリートした命令はグラジュエートできる
2 LW	dec	wait	issue	Address	D Cache	write					
3 ADDI	dec	issue	ALU 2	write	-	-					命令はインオーダーにグラジュエートする
4 SUB	dec	wait	issue	ALU 1	write	-					
5 XORI		dec	issue	ALU 2	write	-					
6 BEQ		dec	wait	wait	issue	ALU 1	(no wrt.)				1サイクルに最大4命令がグラジュエートできる
7 NOP		dec	ready	issue	ALU 1	(no wrt.)	-				
10 AND			dec	issue	ALU 2	write	-				分岐ターゲットキャッシュにヒットすると仮定
11 LB			dec	wait	issue	Address	D Cache	write			

4）グラジュエート（リタイア）

グラジュエートとは物理レジスタの内容を対応する論理レジスタに書き戻す処理である。リタイアとも呼ばれ，インオーダーな完了を実現する。

パイプラインの最終ステージにおいて，命令がコンプリートしていても，すべての依存性が解決され，投機的な実行経路が確定するまでは，グラジュエートできない。R10000は正確な例外を保証するので，例外を起こす命令の後続命令はコンプリートしていても，その命令は同様にグラジュエートできない。

グラジュエート時には，物理レジスタが論理レジスタにリネームし直され，その内容が有効になる。このとき，もっとも前にコンプリートした命令から最初にグラジュエートする。グラジュエートを管理するのはアクティブリストである。あるサイクルにおいて，アクティブリストの先頭から

見て連続してコンプリートしている命令が，その時点でのグラジュエートの対象になる。したがって，アクティブリストの先頭の命令がコンプリートしない限りは1命令もグラジュエートできない。R10000は各サイクルごとに最大4命令をグラジュエートできる。この操作により，命令の流れがその本来のプログラムの順序に戻される。

MIPSの発表ではR10000のIPCは1.5だという。これが，Dhrystone MIPSではなく，本来の意味のMIPS値から求められたものとすればかなりの高性能である。出典は失念したが，4ウェイスーパースカラではIPCが1.5程度が限界だそうである。つまり，R10000は究極の性能を達成しているといえなくもない。

余談であるが，R10000の後継機種であるR12000では，動作周波数が200MHzから300MHzに引き上げられたほかに，マイクロアーキテクチャ的には，アクティブリストが48エントリに，分岐予測テーブルが2048エントリに増加している。また新たに32エントリの分岐ターゲットバッファが追加された。R12000の後継機種としてR14000が開発中であるが，そのアーキテクチャの詳細は，動作周波数が450MHzであること以外は，発表されていない。ロードマップ上はR16000，R18000というMPUも予定されているが，これらが実際に登場するか否かは神のみぞ知るというところであろうか。

●スーパースカラの動作例（R10000の場合）

ここでは，単純なプログラムを用いて，R10000のスーパースカラパイプラインがどのように動作するのかを示す。簡略化のため，整数命令のみを対象とし，論理レジスタは9本（乗除算用のHI/LOレジスタを含む），物理レジスタは16本とする。アクティブリスト，フリーリストはそれぞれ9エントリ，8エントリとする。なお，ここでは整数キューとアドレスキューも共通の命令キューとして考えている。

図16-1にサンプルプログラムと，そのプログラムが実行される場合のパイプラインのタイミングを示す。このプログラムの命令列自体には特別な意味はない。あくまでもパイプライン動作の説明用である。

「dec」は各命令に対するデコードサイクルを意味する。各サイクルで最大4命令がデコード可能である。命令デコードはプログラムの順序（インオーダー）でデコードされる。図示されてはいないが，これらの命令はひとつ前のサイクルで命令キャッシュからフェッチされていることが前提である。また，このサイクルで論理レジスタを物理レジスタに割り付ける作業（レジスタリネーム）が行われる。割り当て可能な物理レジスタ（番号）はフリーリストに登録されており，フリーリストが空になると，不要な物理レジスタが返却されて空でなくなるまで，デコード，レジスタリネームは停止する。レジスタリネーム後は各命令での演

図16-2　サイクル1（以下模式図）

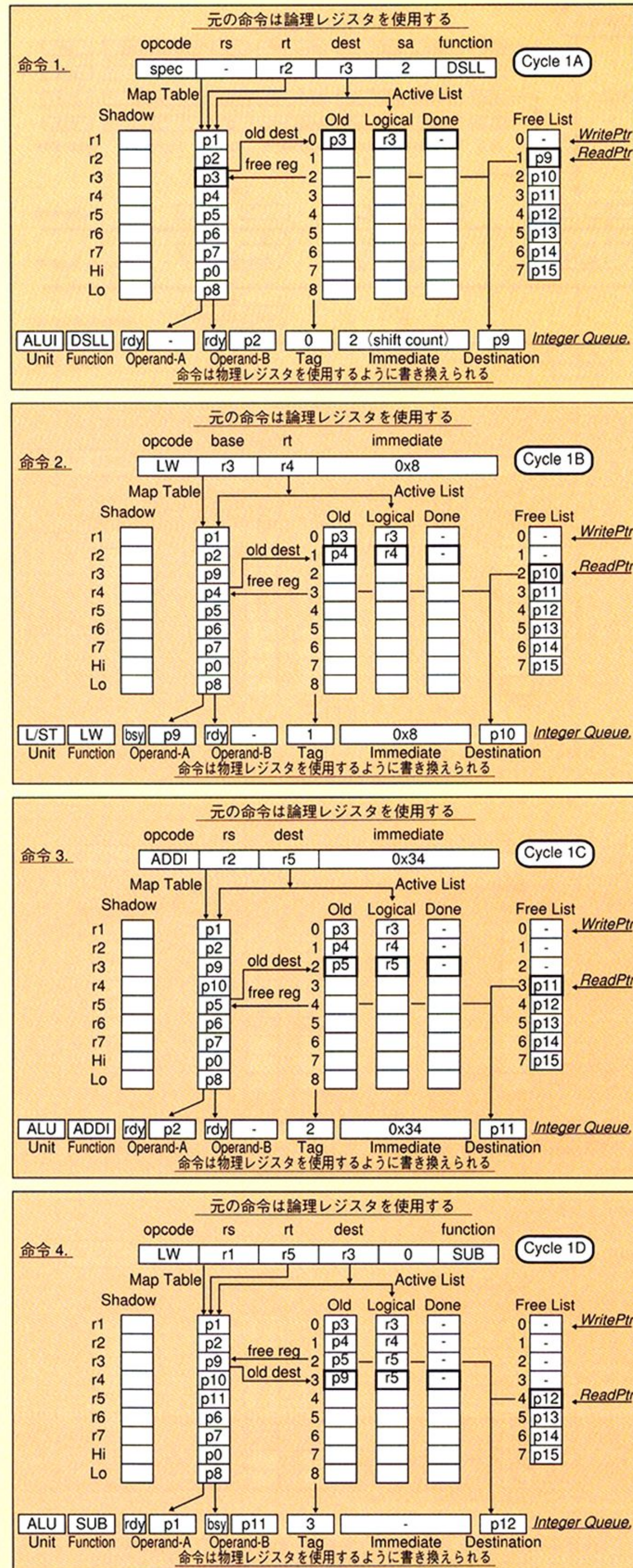
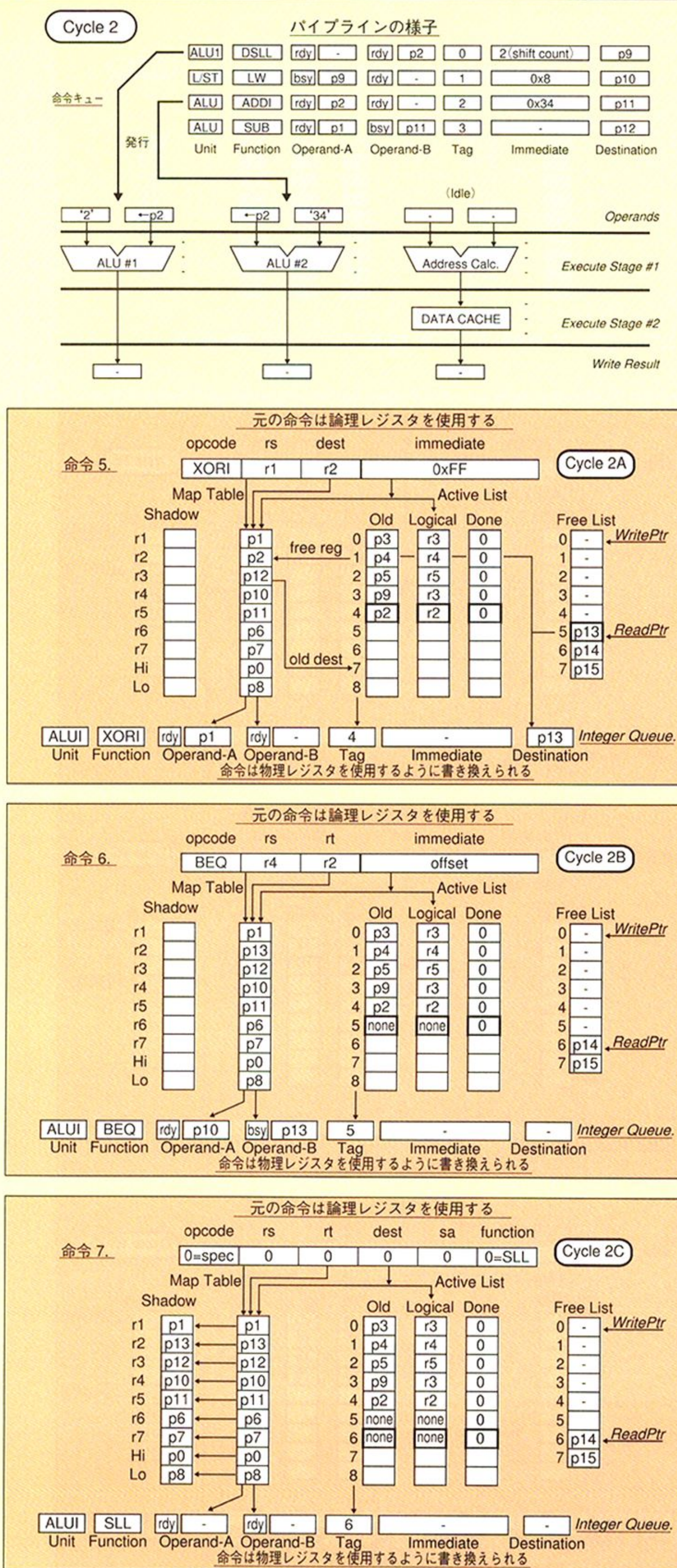



図16-3 サイクル2

算は物理レジスタに対して行われる。

「wait」は命令が実行待ちキューの中で待っていることを示す。具体的には直前の命令のオペランドが使用可能になるのを待っている。

「issue」は命令のすべてのオペランドが使用可能になり，命令が実行ユニットに発行されることを示す。つまり，オペランドが使用可能になった命令からアウトオブオーダーに発行される。命令の実行は次のサイクルで算術論理演算器（ALU1，ALU2）またはロード/ストアを管理するアドレス計算器（Address）によって行われる。

「ready」はすべてのオペランドが使用可能になっているが，その命令を実行する演算器が使用中のため，実行開始が待たされていることを示す。

「write」は物理デスティネーションレジスタへの書き込みを示す。そして，このとき命令の実行が終了（コンプリート）する。さらに，このとき先行する命令がすべてコンプリートしているときに初めて，それらの命令はリタイアできる。つまり，コンプリートはアウトオブオーダーであるが，リタイアはインオーダーである。なお，図中では垂直の太線でリタイアを示している。R10000では1サイクルに最大4命令リタイアできる。命令がリタイアすると，物理レジスタが保持している値は論理レジスタに書き込まれ，物理レジスタは解放されて再利用可能になる（フリーリストに返却される）。

マップテーブルは論理レジスタの数と同じ9エントリを持ち，各エントリは論理レジスタと1対1に対応している。そのエントリはレジスタリネーム作業によって決定され，割り当てられた物理レジスタ番号を保持している。一方，シャドウマップは条件分岐が投機的に実行されるときに使用される。それはマップテーブルをコピーしたもので，もし分岐予測が間違っていたら，その内容がマップテーブルに書き戻される。

アクティブリストはデコードされたが，まだリタイアしていないすべての命令の順序を保持する。いわゆるリオーダーバッファのことである。具体的には論理デスティネーションレジスタと現在すでに割り当てられている物理レジスタの番号（Old）のリストである。「Done」ビットは命令が実行ユニットによってコンプリートしたか否かを示す。

フリーリストはすべての未使用な物理レジスタ番号を保持している。図16-1では物理レジスタはp0からp8がすでに割り当てられた状態（適当にかき混ぜられている）にあり，フリーリストにはp9からp15が残っている。

マップテーブルとフリーリストの初期状態は逐次的にレジスタ番号が割り当てられる。たとえば，r1からr7，HI，LOはそれぞれp0からp8に順次対応し，フリーリストにはp9からp15が残った状態に初期化される。これらの番号は命令が実行されるにつれてかき混ぜられる。

以後，各サイクルにおけるパイプラインの動きを見ていこう。

1）サイクル1（1A，1B，1C，1D）

R10000では通常では1サイクルで最大4命令がデコード/レジスタリネームされる。ここでは，命令1，命令2，命令3，命令4が同時にデコードされ命令キューに格納される。この様子を図16-2に示す。

命令1 (DSLL)：Doubleword Shift Left Logical
ソースオペランドであるrtは，すでに論理レジスタr2から物理レジスタp2に割り当てられている。その論理デスティネーションレジスタであるr3はフリーリストからp9が選択され新しい物理レジスタとして割り当てられる。すでに割り当てられていたp3はアクティブリストのOld領域に書き込まれる。なお，この命令（シフト命令）はALU1のみで実行可能である。

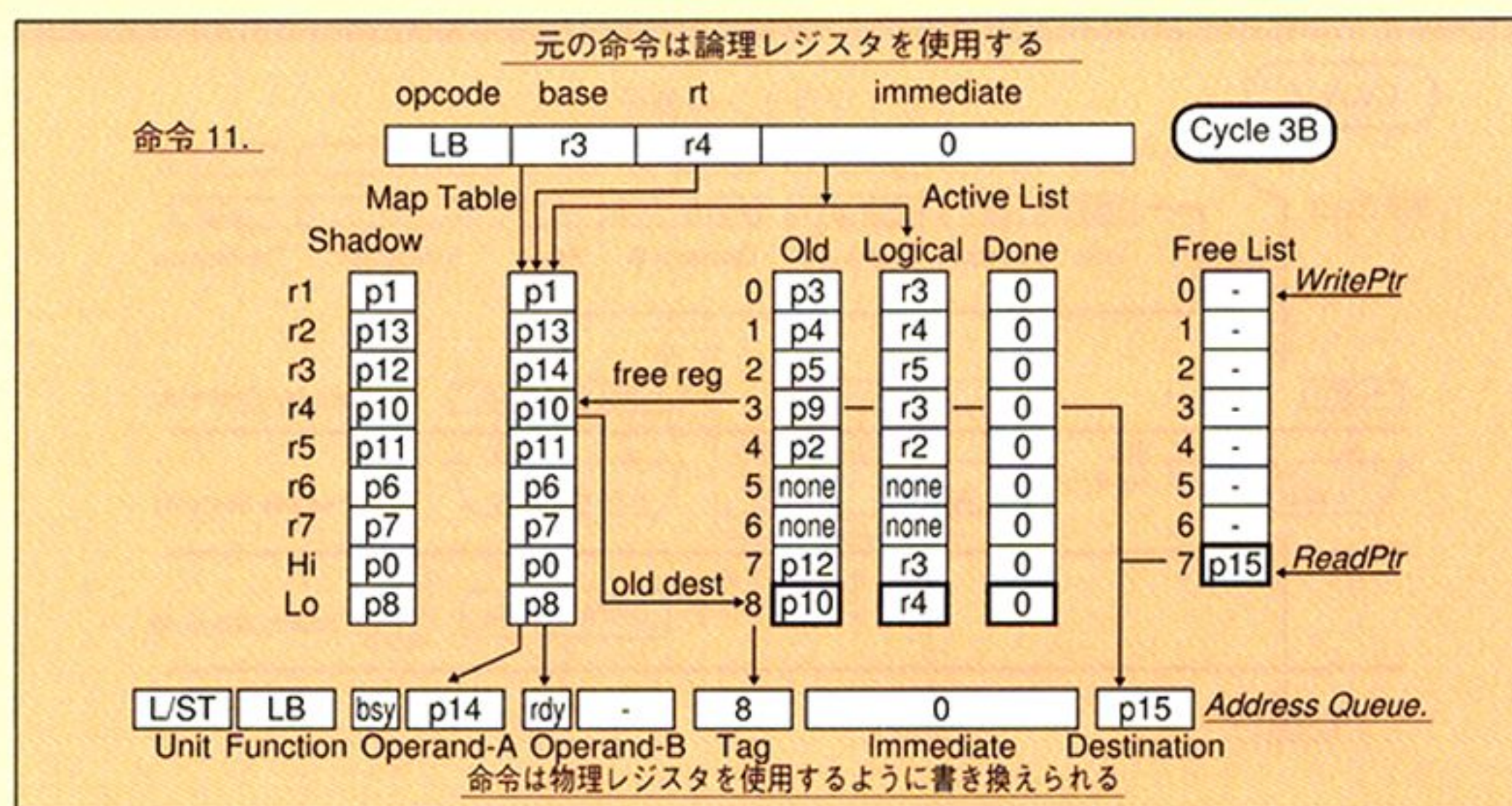
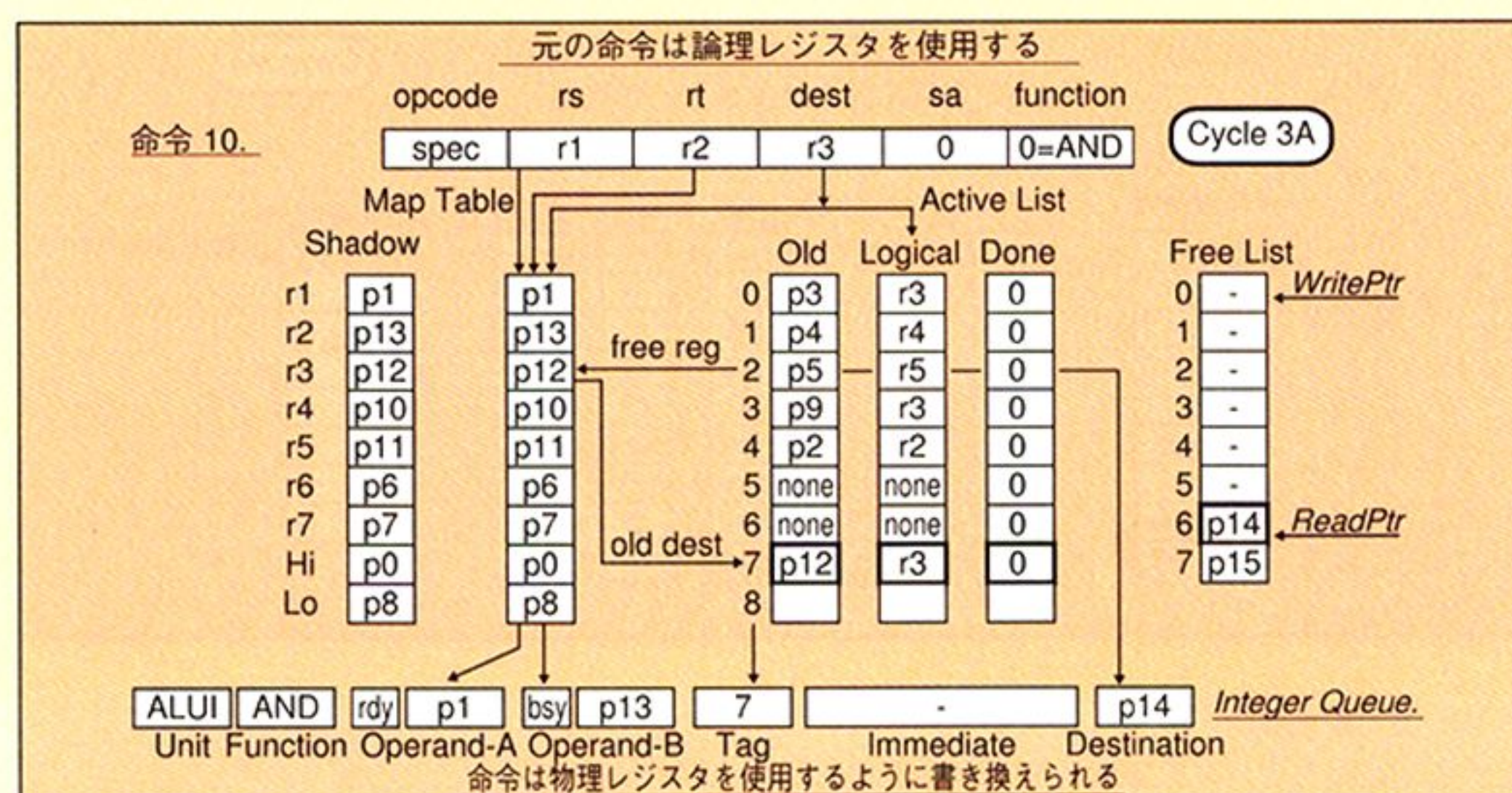
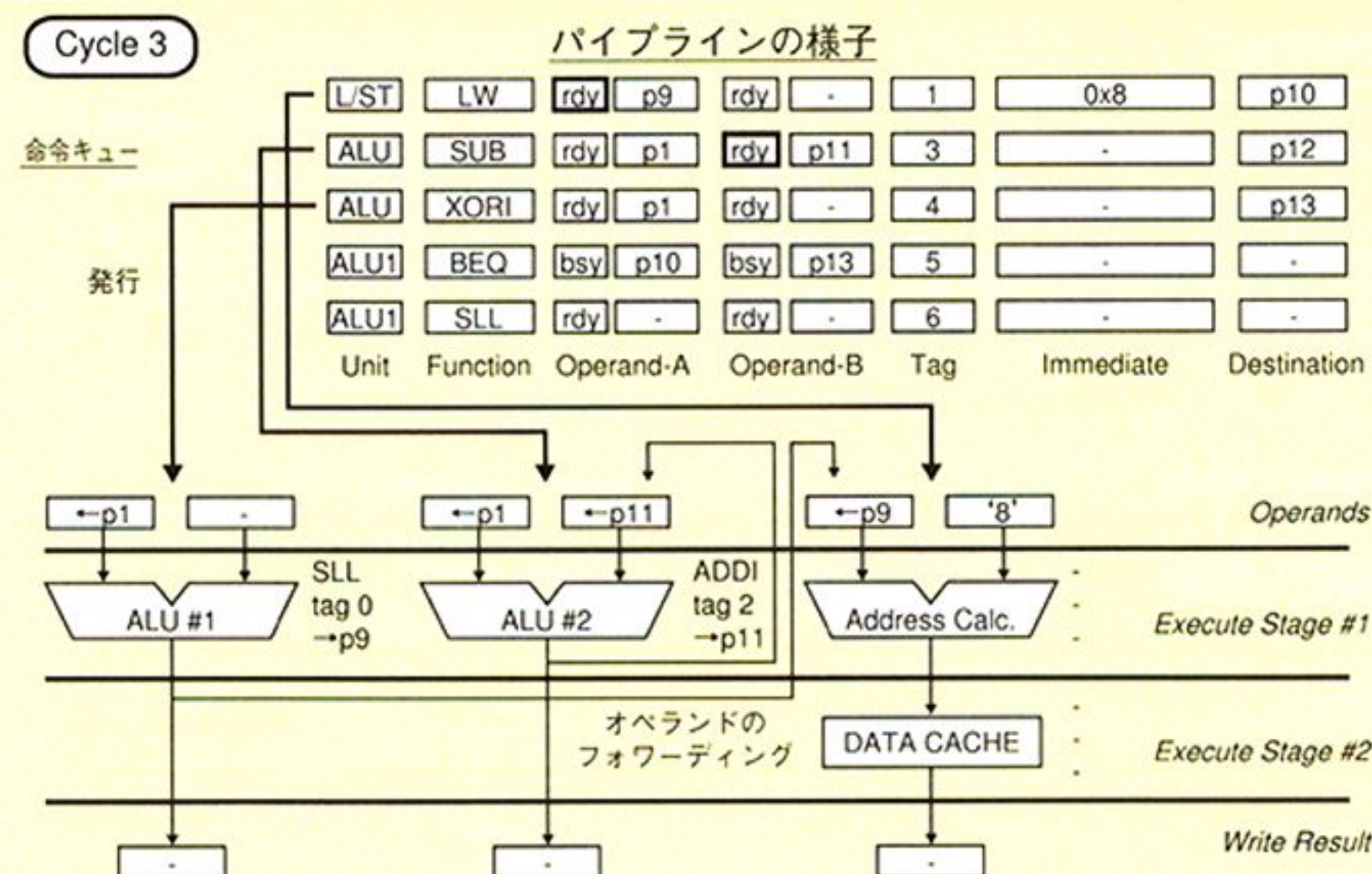
命令2 (LW)：Load Word
アドレス計算のためのベースレジスタを保持するbaseはr3なのでp9へ割り当てられている。このレジスタは先行する命令のデスティネーションレジスタであり，依存性を持っている。このためp9は命令キューの中ではビジー（bsy）とマークされる。rt (r4) は論理デスティネーションレジスタである。これはフリーリストから新しい物理レジスタp10へ割り当てられる。古い割り当てであるp4はアクティブリストに書き込まれる。この命令（ロード命令）はロード/ストアユニットのみで実行可能である。

命令3 (ADDI)：Add Immediate
イミディエート値に加算されるソースオペランドであるr2はp2に割り当てられている。このレジスタは先行する命令と依存性がなく，命令キューの中ではレディ（rdy）とマークされる。rt (r5) は論理デスティネーションレジスタである。これはフリーリストから新しい物理レジスタp11に割り当てられる。その古い割り当てであるp5はアクティブリストに書き込まれる。この命令（加算）はALU1またはALU2で実行可能である。

命令4 (SUB)：Subtract
除算命令は2つのソースオペランドを持つ。片方のr1はp1に割り当てられており，依存性はない（rdy）。もう片方のr5はp11に割り当てられており，これは先行する命令に依存する（bsy）。論理デスティネーションレジスタr3はフリーリストから新しい物理レジスタp12に割り当てられる。このレジスタは最初の命令（DSLL）と出力依存性を持つが，これはレジスタリネームによって解消されている。なお，古い割り当てであるp9はアクティブリストに書き込まれる。この命令（減算）はALU1またはALU2で実行可能である。

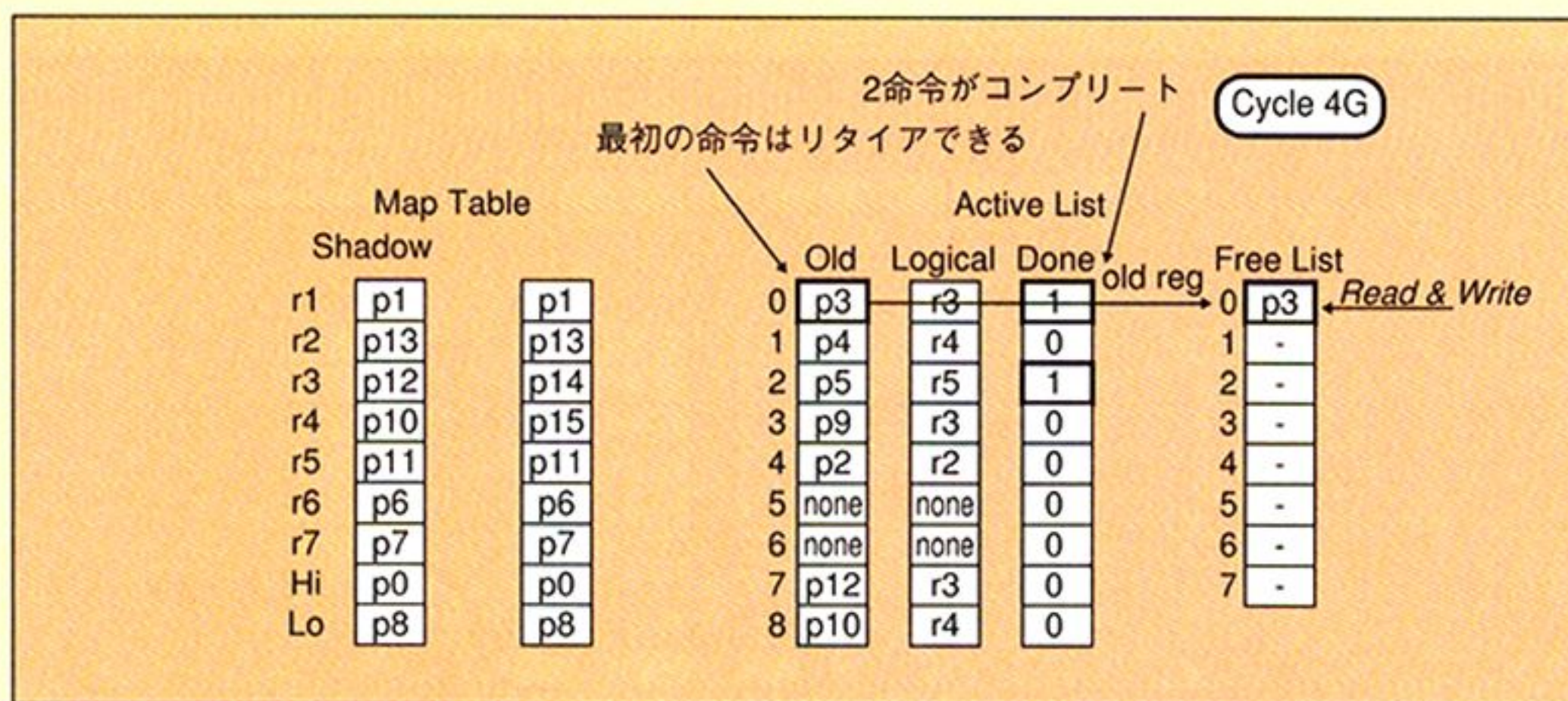
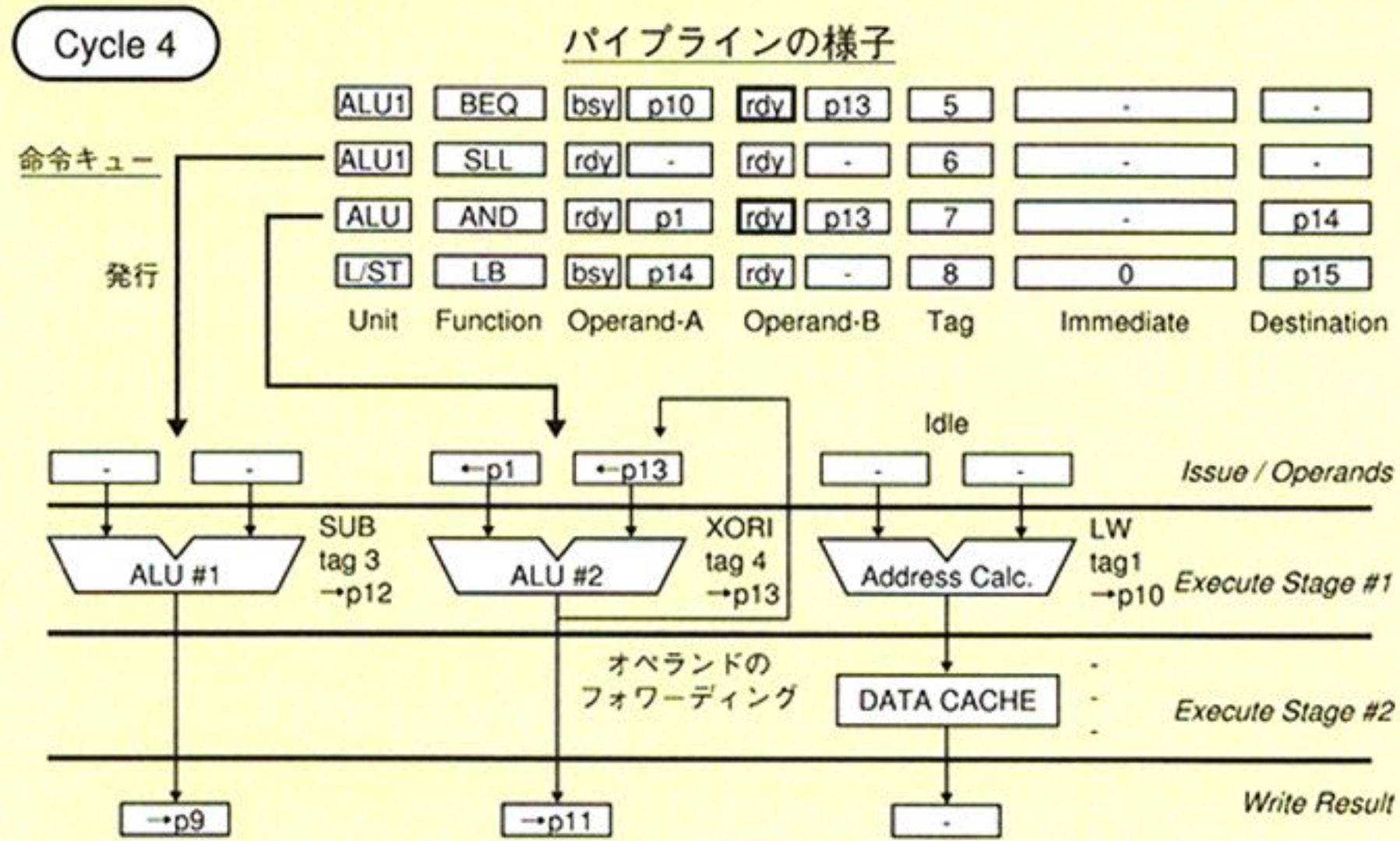
図16-4 サイクル3

2)サイクル2 (2A，2B，2C)

4命令がサイクル1の間にデコードされた。サイクル2の間では，これら4命令は命令キューの中にあり，整数ALUやロード/ストアユニットに対して発行が可能である。図16-3に示すように，すべてのソースオペランド（Operand-AとOperand-B）が両方レディになっている，Tag（命令につけられた番号）が0と2の命令は発行可能である。Tag0はALU1に発行される。Tag2はALU1とALU2の両方に発行可能であるが，いまは（空いている）ALU2に発行される。これらの命令に対するレジスタオペランドはサイクル2の後半でレジスタファイルからリードされる。そして，これらの命令はサイクル2の終わりに命令キューから削除される。他の2命令はソースオペランドが有効（レディ）になるまで待っている。

同時に，サイクル2では新たに3命令がデコード/レジスタリネームされる。本来は4命令をデコード可能であるが，4命令内に条件分岐命令（BEQ）を含むため，その遅延スロットまでしかデコードされない。

命令5 (XORI)：Exclusive Or Immediate
イミディエート値と排他的論理和がとられるソースオペランドであるr1はp1に割り当てられて

図16-5 サイクル4

図16-6 サイクル5

いる。このレジスタには依存性はない。論理デスティネーションレジスタであるr2はフリーリストから物理レジスタp13に割り当てられる。そして，古い割り当てであるp2はアクティブリストに書き込まれる。この命令（排他的論理和）はALU1とALU2の両方で実行可能である。

命令6（BEQ）：Branch On Equal

この命令は2つのレジスタオペランドを比較し，それらが等しいと分岐する。比較は数サイクル経過しないと完了できないので，R10000は分岐予測を行う。この例では分岐は成立すると予測している。1番目のソースオペランドr4はp10に割り当てられている。2番目のソースオペランドr2はp13に割り当てられている。どちらのレジスタも先行する命令の結果に依存しているため，命令キューの中ではビジーにマークされる。この命令はデスティネーションレジスタを持たないのでフリーリストから新たな物理レジスタを割り当てる必要はない。この命令（条件分岐）はALU1のみで実行可能である。

命令7（NOP=SLL）：No-Operation

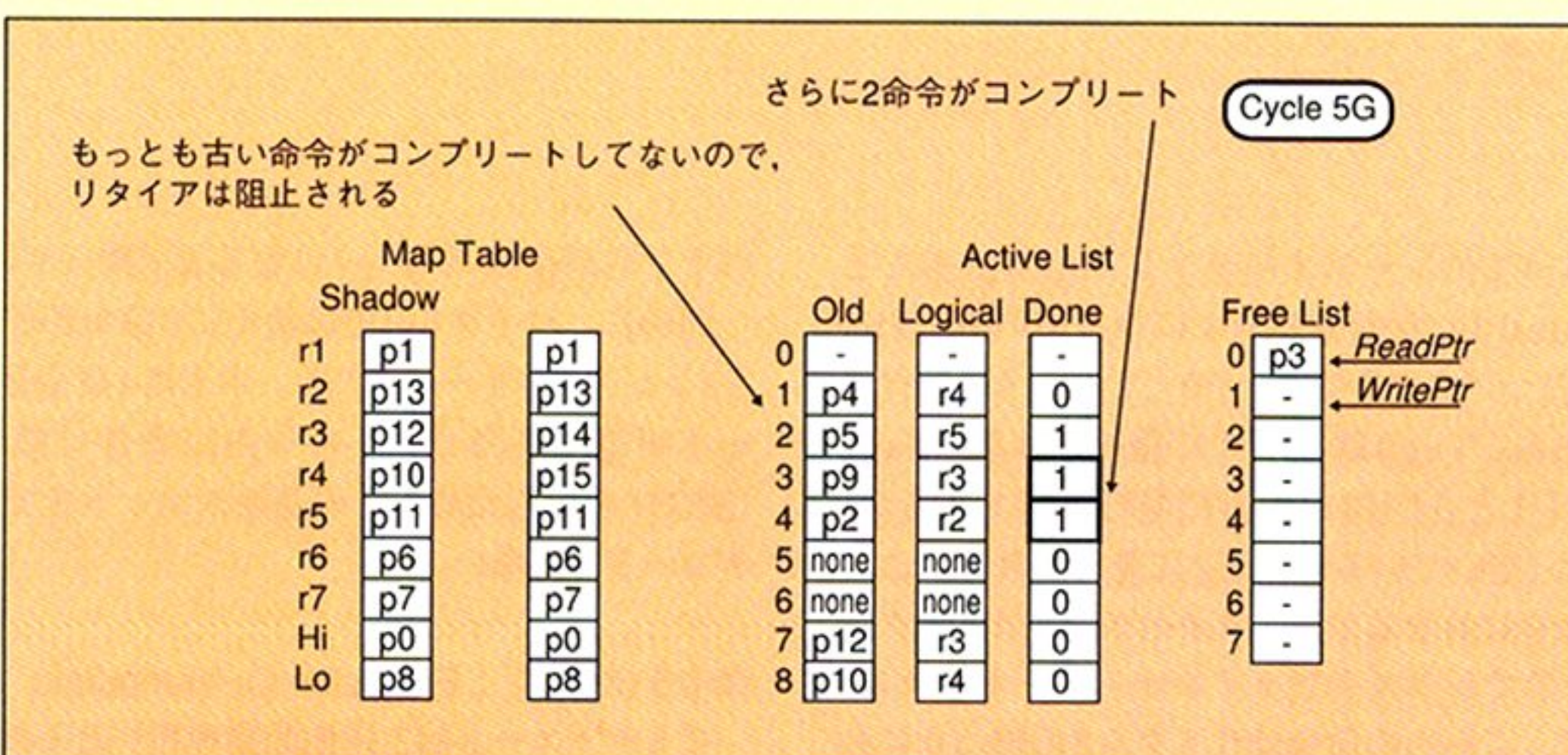
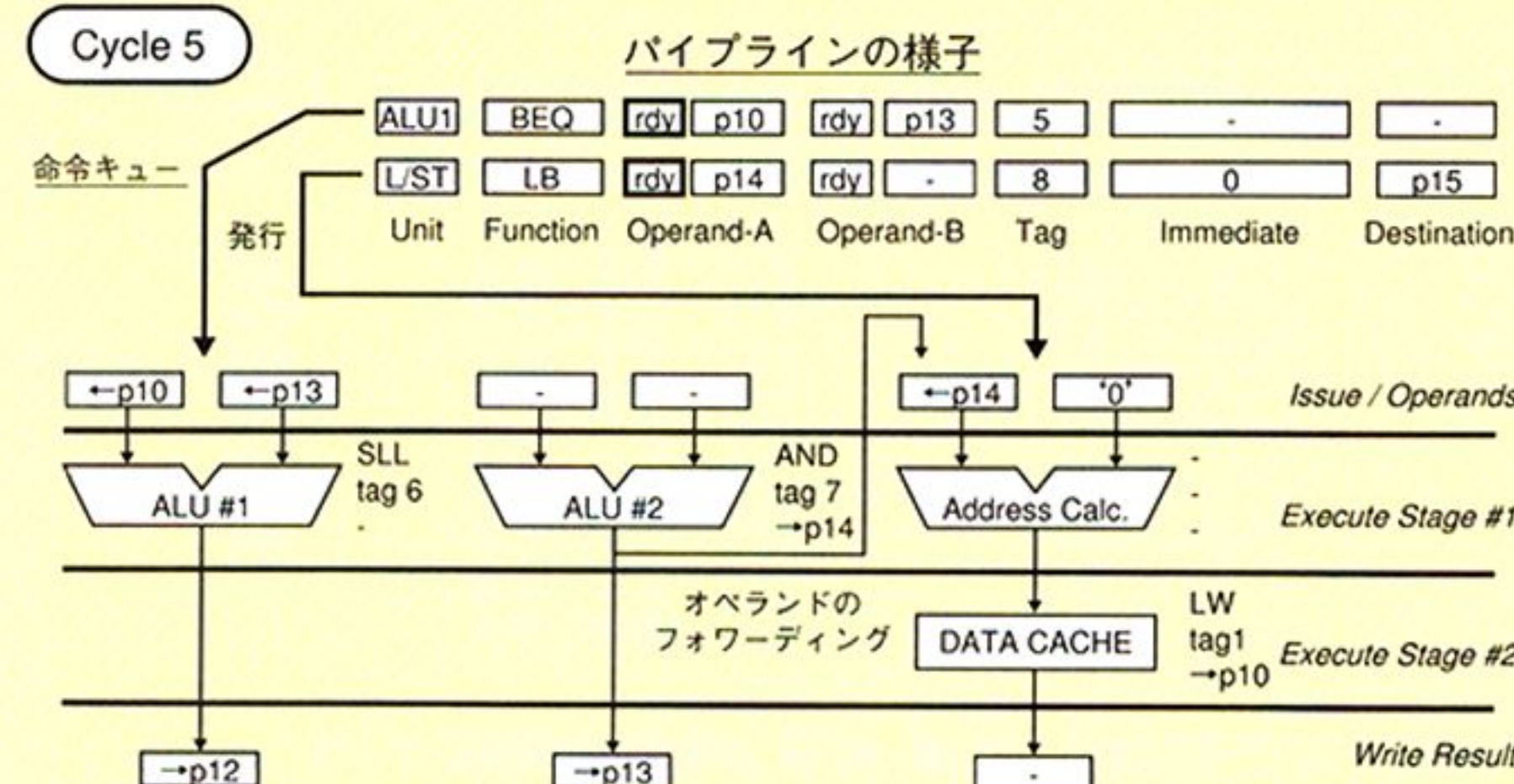
これはR10000の状態になんの影響も与えないアセンブラの疑似命令である。実体はシフト量0のシフト命令（SLL）である。そのソースオペランドはゼロレジスタであり，値として0が使用される。デスティネーションレジスタもゼロレジスタであり，この場合は結果の書き込みは行われない。結果としてNOPには依存性は存在しない。この命令（シフト）はALU1のみで実行可能である。

この例ではNOPはシフト命令のままであるが，現実の実装では命令キャッシュに書き込む時点で，NOPはすべてのオペランドがゼロレジスタの加算（ADD）命令にプリデコードされる。これにより，NOPはALU1とALU2の両方で実行可能になっている。

遅延スロットの命令は分岐が実際に行われる前に実行される。直前のBEQは条件分岐命令なので，それは分岐予測回路に基づいて投機的に実行される。この予測は後に正しいか否かが判定される。条件分岐が分岐すると予測された場合，マップテーブルの内容がシャドウマップにコピーされる。マップテーブル自身は分岐先の命令が投機的にデコード/実行されるにつれて更新されていく。もし予測が間違っていたらマップテーブルはシャドウマップから再ストアされる。これは遅延スロットの実行時点まで，命令の実行を巻き戻すことを意味する。

BEQLのようなBranch Likely命令は遅延スロットの前にマップテーブルをコピーする。なぜなら，もし分岐が成立しないなら遅延スロットの命令は実行されないためである。

3）サイクル3（3A，3B）

図16-4に示すように，命令キューはサイクル3の開始時点で5命令を含んでいる。そのうち，2

命令は前のサイクルからの残りで，3命令はサイクル2でデコードされたものである。サイクル2の間に発行された2命令（Tag0とTag2）はどちらも1サイクルのレイテンシなので，その物理デスティネーションレジスタであるp9とp11はもはやビジーではない。値をほかのレジスタにフォワーディングすることができるためである。このため，これまでビジーだったレジスタがレディとなり，Tagが1，3，4，6の4命令が発行可能になる。Tag1はアドレス計算ユニットに発行される。Tag3はALU1に発行される。Tag4はALU2に発行される。Tag6もレディであるが，このサイクルでは発行できない。なぜなら，ほかの命令がすでに両方のALUに発行されているためである。Tag1，Tag3，Tag4の命令のレジスタオペランドはサイクル3の後半にレジスタファイルからリードされる。また，発行された命令はこのサイクルの終わりにキューから削除される。ほかの2命令はオペランドが有効になるのを待っている。

サイクル3では（予測した分岐先の）2命令をデコード/レジスタリネーム可能である。本来なら4命令をデコード可能であるが，アクティブリストの空きが2命令分しかない（あるいはフリーリストにレジスタが2つしか残っていない）ため，2命令に制限されている。

命令10（AND）：Logical AND

片方のソースオペランドであるr1はp1に割り当てられていてレディである。もう一方のソースオペランドであるr2はp13に割り当てられている。これは命令5（XORI）のデスティネーションレジスタに依存しているためビジーである。論理デスティネーションレジスタr3はフリーリストから新しい物理レジスタp14が割り当てられる。それ以前の割り当てであるp12はアクティブリストに書き込まれる。この命令（AND）はALU1とALU2で実行可能である。

命令11（LB）：Load Byte

アドレス計算のためのベースレジスタであるr3はp14に割り当てられている。これは先行する命令と依存性がある。論理デスティネーションレジスタであるr4はフリーリストから新しい物理レジスタp15に割り当てられる。古い割り当てであるp10はアクティブリストに書き込まれる。この命令（LB）はロード/ストアユニットで実行可能である。

4）サイクル4（4G）

図16-5に示されるように，サイクル4の最初では命令キューは4命令を含んでいる。そのうち，2命令は前のサイクルからの残りで，2命令はサイクル3の間にデコードされたものである。サイクル3の間に発行された2つのALU命令（Tag3とTag4）はそれぞれ1サイクルのレイテンシを持っている。このため，その物理デスティネーションレジスタであるp12とp13はもはやビジーでない。しかし，ロード命令（Tag1）は2サイクルのレイテンシである（AddressステージとDCacheステージ）。そのため，p10はビジーのままである。Tag6，Tag7はサイクル4で発行する準備ができた。Tag6はALU1に，Tag7はALU2に発行される。これらの命令のレジスタオペランドはサイクル4の後半にレジスタファイルからリードされる。発行された命令はこのサイクルの終わりで命令キューから削除される。ほかの2命令はそれらのオペランドが有効になるまで待っている。

このサイクルではアクティブリストに空きがないため新たな命令のデコードは発生しない。

このサイクルでは2つのALU演算の結果がレジスタファイルに書き込まれる。そして，対応する2命令（Tag0とTag2）はアクティブリストの中でコンプリート（Done）という印がつけられる。このとき，最初の命令（Tag0，DSLL）はサイクル4の終わりでリタイアできる。命令3（Tag2，ADDI）はリタイアできない。なぜなら2番目の命令（Tag1）がコンプリートしていないためである。同時にリタイアできる命令はアクティブリストの中で連続してコンプリートしている最大4命令である。リタイアした命令のデスティネーションレジスタであるp3は再利用できるようにフリーリストに返される。同時に対応するアクティブリストのエントリも空になり再利用することができるようになる。

5）サイクル5（5G）

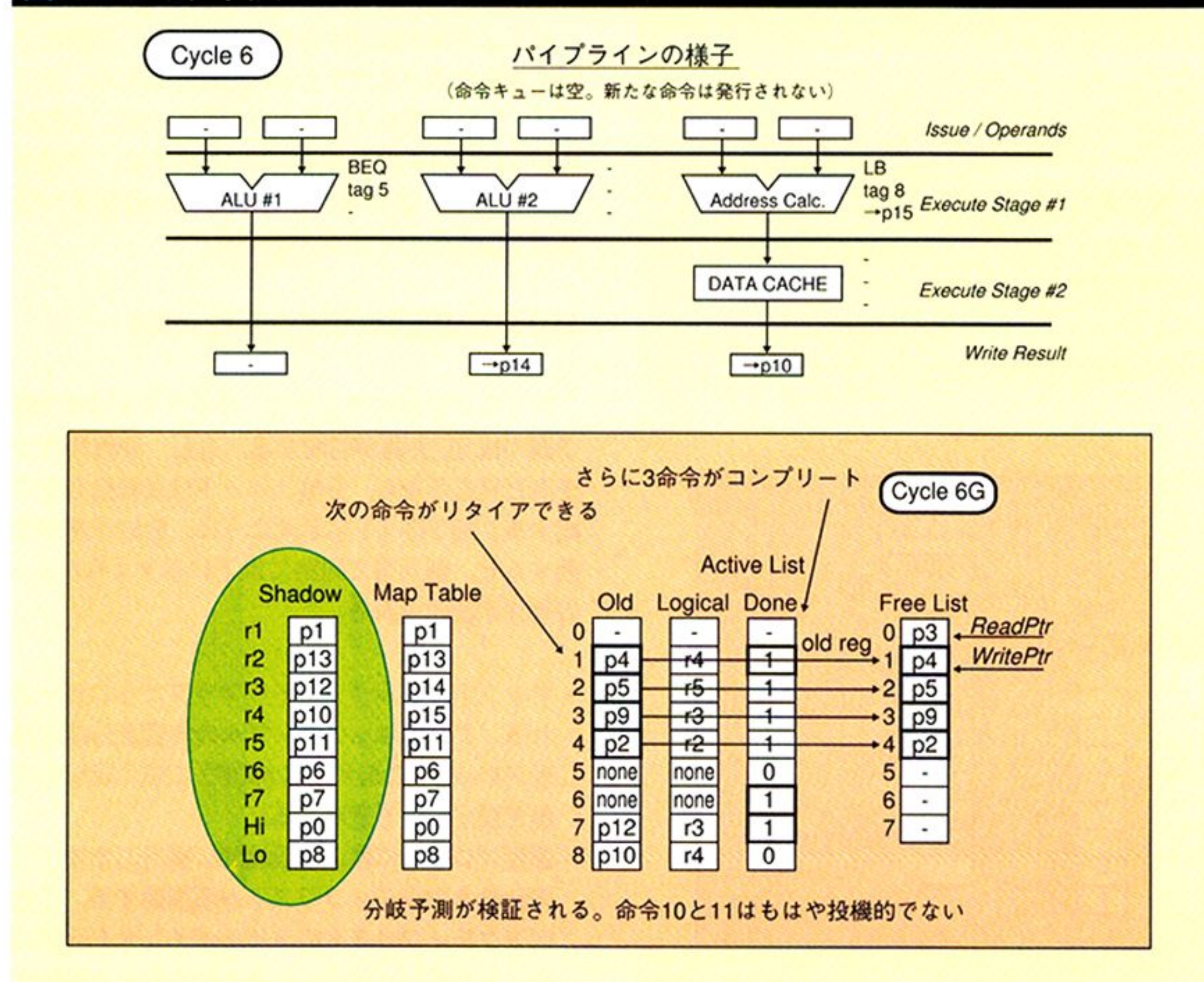
図16-6に示すように，サイクル5では命令キューは先のサイクルから残っている2命令を含んでいる。サイクル4の間に発行された2命令（Tag6とTag7）はそれぞれ1サイクルのレイテンシを持っている。そのため，Tag7の物理デスティネーションレジスタであるp14はもはやビジーではない。Tag6はデスティネーションレジスタを持たないNOPである。命令1（Tag0，LW）はこのサイクルでその2サイクルのレイテンシを完了する。このため，そのデスティネーションレジスタであるp10がレディになる。サイクル5ではTag5とTag8の命令が発行可能になる。Tag5はALU1に発行され，Tag8はロード/ストアユニットに発行される。これらの命令のレジスタオペランドはサイクル5の後半にレジスタファイルからリードされる。発行された命令はこのサイクルの終わりで命令キューから削除される。

このサイクルでは，アクティブリストに空きがあるため，さらに1命令をデコード可能であるが，この例ではそこまでの動作は示していない。

サイクル5では2つの命令（Tag3とTag4）がコンプリートする。そして，アクティブリストの対応するエントリにDoneの印がつけられる。このサイクルでもTag1の命令がコンプリートしていないため，どの命令もリタイアできない。

6）サイクル6（6G）

図16-7に示すように，サイクル6では命令キューは空である。このサイクルではTag5の分岐命令（BEQ）の結果が判明する。さらに，AND命令（Tag7）の結果がレジスタファイルに書き込まれる。またLB命令（Tag8）のためにアドレス

図16-7 サイクル6

計算が行われる。先のLW命令(Tag1)は結果をレジスタファイルに書き込む。サイクル6ではTag1, Tag6, Tag7がコンプリートし，対応するアクティブリストのエントリにDoneの印がつけられる。Tag1がコンプリートすることにより，アクティブリストの1から4に対応する連続4命令がコンプリートしたことになり，この4命令がリタイアできる。また，対応するOld領域に格納されていた物理レジスタ(p4, p5, p9, p2)がフリーリストに返される。

この例では分岐予測が成功したと仮定している。このとき，命令10, 11(Tag7, Tag8)はもはや投機的ではなくなる。ここまで変更してきたマップテーブルは有効ということになり，シャドウマップの値は不要になる。

図16-8 サイクル7

図16-9 サイクル8

7) サイクル7(7G)

図16-8に示すように，サイクル7ではLB命令(Tag8)によるデータキャッシュへのアクセスが行われる。また分岐命令(Tag5)にコンプリートし，アクティブリストのエントリがDoneの印がつけられる。このとき，アクティブリストの中で分岐命令がもっとも古く，その遅延スロット(Tag6)と分岐先(Tag7, AND)も既にコンプリートしているので，その3命令が同時にリタイアする。分岐とNOP(遅延スロット)はデスティネーションレジスタを持たないため，古い物理レジスタをフリーリストに返すことはない。AND(Tag7)はp12をフリーリストに返す。

8) サイクル8(8G)

図16-9に示すように，サイクル8ではLB命令(Tag8)の結果がレジスタファイルに書き込まれる。また，同時にアクティブリストにはDoneの印がつけられ，LB命令がもっとも古い命令であるため，リタイアする。古いデスティネーションレジスタのp10はフリーリストに返される。

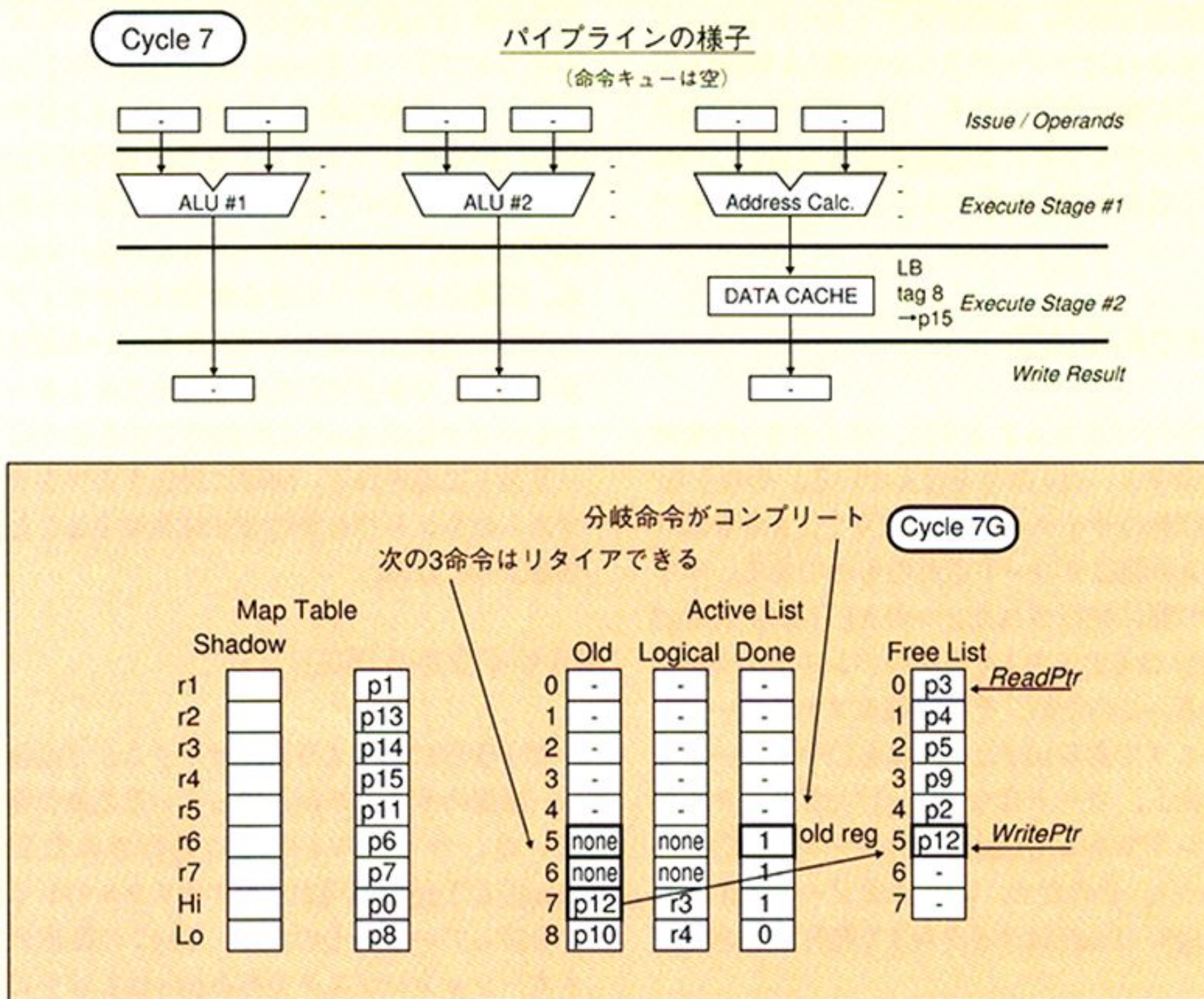
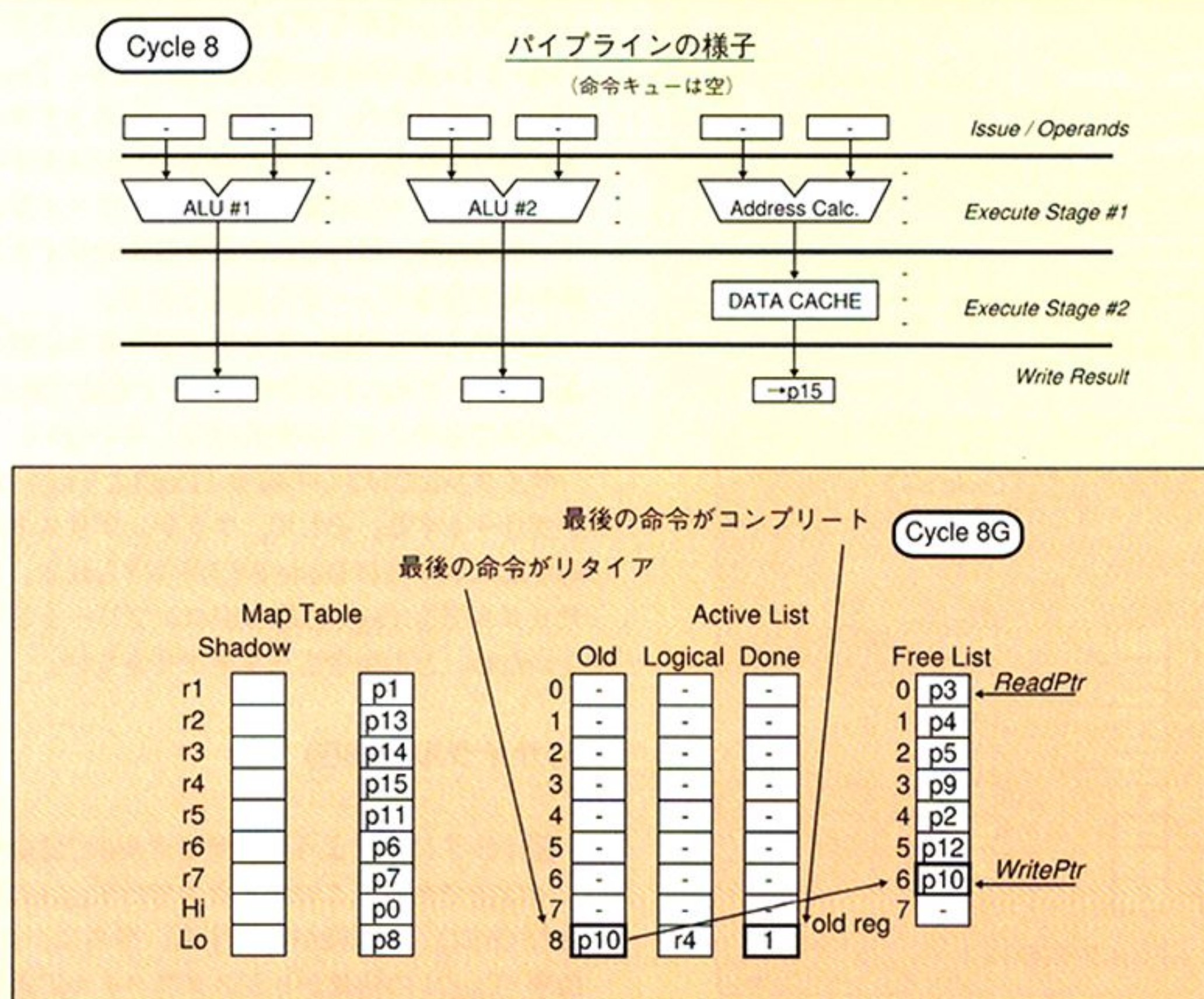
この時点で，パイプラインは空になり，アクティブリストも空になる。フリーリストには再び7個のレジスタが存在する。マップテーブルは現在の論理レジスタと物理レジスタの割り当て関係を示している。

9) 分岐予測が外れる場合

以上は分岐が成立するものと予測し，実際に分岐する場合のパイプラインの動作を示した。ここでは，分岐が成立するものと予測したが，分岐しなかった場合の例を示す。図16-10に，分岐予測が失敗し，実行処理を分岐命令の直後まで巻き戻す場合のタイミングを示す。

10) 分岐の反転(サイクル6, 6G)

図16-7に示したように，サイクル6では分岐予測の成功/失敗が判明する。もし，分岐条件が予測と異なるなら，分岐ユニットは反転信号を生成する。図16-11に示すように，分岐予測が失敗すると，割り当て回路に以下に示すような3つの動作が要求される。

- マップテーブルがシャドウマップからコピーされる。これはマップテーブルの内容を分岐の遅延スロット(この例ではNOP)に続く状態まで巻き戻すことを意味する。
- 遅延スロットに続いて投機的に実行したすべての命令をアクティブリストから削除する。これはアクティブリストのライトポインタを遅延スロットがデコードされたあとの位置まで移動す

ることで実現する。

・これらのアボートされた命令で使用された物理レジスタ(p14とp15)をフリーリストに返す。これはフリーリストのリードポインタを巻き戻すことで実現する。明確化のために，図中では空白で示されているが，フリーリストからレジスタが割り当てられたあとも，エントリの値が消去されるわけではなく，エントリ6，7にはp14，p15が残ったままになっているので，リードポインタの操作だけで物理レジスタを回復できる。

11)分岐の反転(サイクル7，7R)

図16-12に，分岐予測の失敗後の動作を示す。基本的に4命令をデコードできるが，乗算命令があると，アクティブリストを特殊な使い方をするためデコードが中断する。

命令8(MULT)：Multiply

乗算命令は2つの特殊目的の論理デスティネーションレジスタであるHIとLOを持つ。これらは通常の論理レジスタと同様に扱われ，フリーリストから2つの物理レジスタが割り当てられる。しかし，R10000では，アクティブリストの各エントリはただひとつのデスティネーションレジスタしか収容できないので，この場合，連続する2つのエントリが使用される。2番目のエントリは次の命令用なので，このサイクルではこれ以上の命令をデコードできない。そして，この2番目のエントリにはあらかじめデコード中にDoneの印がつけられる。これにより，1番目のエントリがリタイアすると同時に，2番目のエントリもリタイアできる。

乗算は完了するのに数サイクルかかる。そのため，その結果は引き続く命令でただちに利用可能とはならない。つまり，次のサイクルで乗算結果を取り出すMFLO命令がデコードされるが，そのソースオペランド(LO=p15)はすぐにはレディにならない。

サイクル7では，図16-8と同様に，分岐命令(Tag5)がコンプリートし，その分岐命令と遅延スロット(Tag6)がリタイアする。

●おわりに

スーパースカラの概要について解説してきた。文章の量はかなり多くなったが内容的にはそれほど難しいことはいっていない。MPUの複数の命令の(特にアウトオブオーダーな)同時実行の方式をR10000をモデルに詳しく説明したが理解できたであろうか。最近のMPUはアウトオブオーダーなスーパースカラが常識のようになってきているので，スーパースカラの基礎を押さえておくことは必須であろう。次章では並列処理のもうひとつの手法であるVLIWについて解説する。

図16-10 分岐予測失敗時のサンプルプログラム

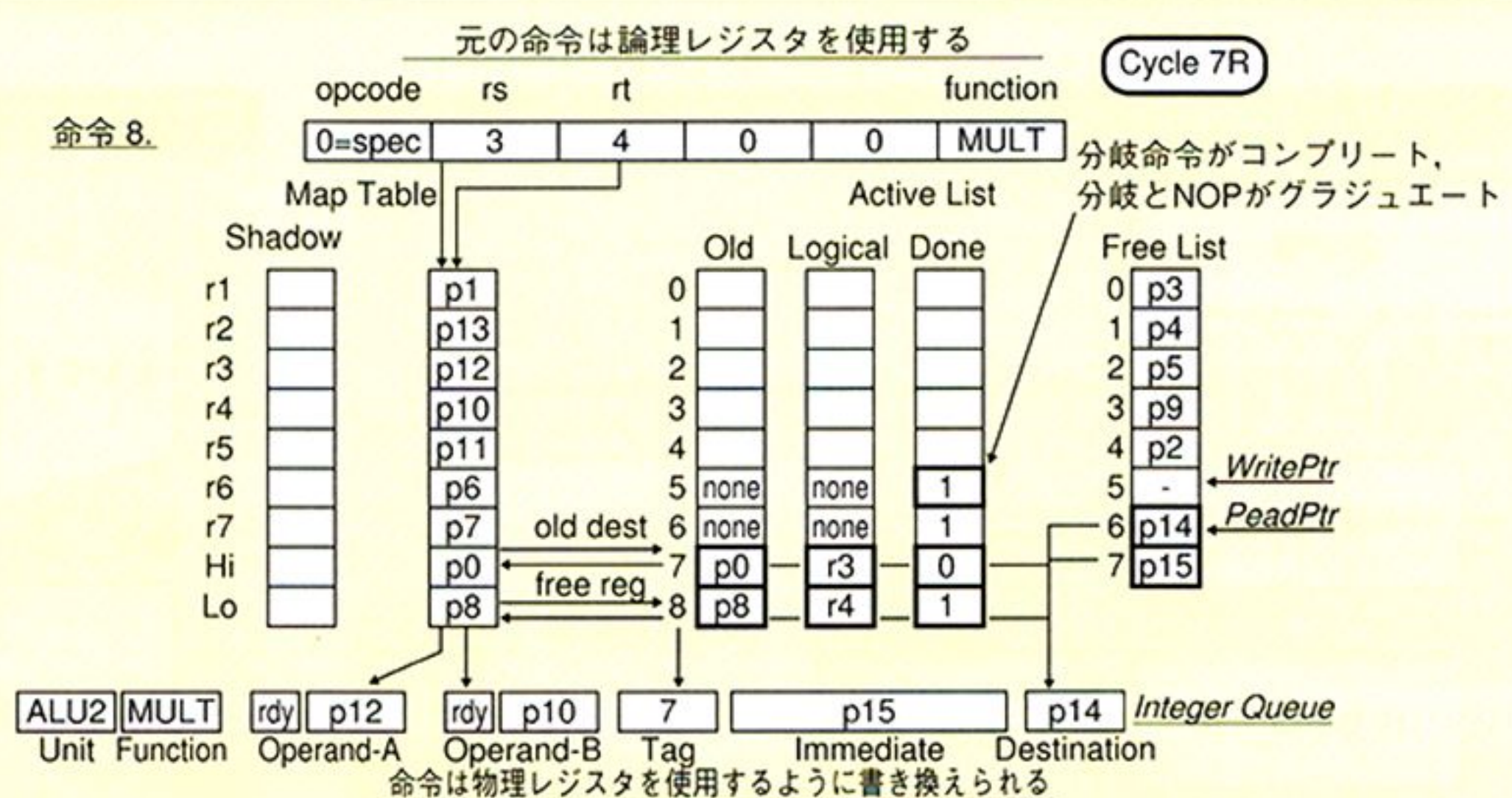
アセンブラのコード		実際の演算	リネーム	使用ユニット	サイクル	備考
…						
6 BEQ	r4,r2,label	branch if p10=p13	none	ALU1	2B	条件分岐(TAKENと予測)
7 NOP	-	no operation	none	ALU	2C	遅延スロット
8 MULT	r3,r4	p14,p15←p12*p10	Hi=p14 Lo=p15	ALU2	7R	分岐反転後にデコード 乗算はデスティネーションレジスタを2個持つ
9 MFLO	r2	…				
…						
label						
10 AND	r3,r1,r2	p14←p1 and p13	r3=p14	ALU	3A	投機的にデコードされるがアボートされる
11 LB	r4,0(r3)	p15←mem(p14+0)	r4=p15	L/ST	3B	

パイプライン タイミング

命令	1	2	3	4	5	6	7	8	9	10
6 BEQ		dec	wait	wait	issue	ALU 1	(no wrt.)			
7 NOP		dec	ready	issue	ALU 1	(no wrt.)	-			
10 AND			dec	issue	ALU 2	write	Abort!			
11 LB			dec	wait	issue	Address	Abort!			
8 MULT						reverse	dec	issue	ALU 2	ALU 2 …
9 MFLO								dec	wait	wait …

分岐予測はALU 1で検証され誤りがあることが判明する。投機的に実行した命令はアボートする。もう一方の経路(NOT TAKEN)がフェッチされデコードされる。Branch Reverse Cacheは次の4命令を保持する(そうでなければ，デコードの前にさらなるフェッチサイクルが必要)

図16-11 サイクル6(分岐予測失敗)

図16-12 サイクル7(分岐予測失敗)

Computer Architecture

コンピュータアーキテクチャ

その直感的アプローチ 5

マイクロプログラミングとVLIW

中森 章 *Nakamori Akira*

今回はマイクロプログラミングとVLIWを取り上げる。マイクロプログラミングは最近では見られなくなったMPUの実現方式であるが，あえて取り上げるのには理由がある。それはVLIWとの関係である。どのような関係があるのか，それは本稿を読んでもらえばわかると思う。いまどき必要ないじゃん，といわずに読んでみてほしい。

●マイクロプログラミングとは

マイクロプログラミングの概念は1951年にケンブリッジ大学のM.V.Wilkesによって提案された。その目的はコンピュータの制御系を効率的に設計することである。

図1にコンピュータ（MPU）の基本的な構成図を示す。プログラム（命令）は主記憶（キャッシュの場合もあるが）からバス制御ユニットによって読み出され，命令デコーダでデコードされる。そして，そのデコード情報をもとに命令が実行（処理）される。この実行制御部の設計を効率的に行えるのがマイクロプログラミングである。

MPUの実行はマイクロ操作（micro-operation）と呼ばれる基本操作の組み合わせによって実現される。制御回路はマイクロ操作を制御する信号を次々と発生する。たとえば，図2にMPUの実行制御部の概念図を示す。デコードした結果をもとに，レジスタ番号を指定する（①），演算の種類を指定する（②），レジスタを選択する（③），レジスタの内容を演算器に与える（④），演算結果をレジスタにライトバックする（⑤）といった操作を実行することで命令が実行できるが，これらの操作がマイクロ操作である。実行初期のMPUや最近のRISCでは，論理回路の組み合わせでこの制御信号を生成する。これがワイアードロジック（wired logic）と呼ばれる方式である。ワイアードロジックの設計は非常に複雑であり，設計ミスをした場合の修正も容易ではない。

マイクロ操作を実現するためのマイクロ命令というものを考える。つまり，マイクロ命令は同時に発生する制御信号に関する情報をまとめて命令の形式に収める。こうすることで，MPUの実行はマイクロ命令の系列として定義できる。一般にMPUのプログラムは命令が並んだものであるが，その個々の命令の処理は対応するマイクロ命令を実行する（複数のマイクロ操作を同時に行う）ことで実現できる。これをマイクロプログラムと呼ぶ。マイクロプログラムはマイクロコードと呼ばれることもある。意味的には，ハードウェアとソフトウェアの中間にあることからファーム（やや硬い）ウェア（firmware）と呼ばれることもある。また，マイクロプログラムでMPUの実行を制御する方式を，マイクロプログラム制御方式，あるいはマイクロプログラミング方式と呼ぶ。なお，マイクロ命令に対応して，通常の命令はマクロ命令と呼ばれる。

マイクロプログラム制御方式において，マイクロプログラムは適当な記憶装置（通常はROMなので，これをマイクロROMと呼ぶ）に格納される。そして，マクロ命令はマイクロROM内のアドレスがエンコードして格納されているとみなすことができる。つまり，あるマクロ命令がデコードされるとマイクロROMのアドレスが決定される。同時に，オペランドの種類，演算の種類がデコードされて保持される。命令の実行ステージでは，そのマイクロROMからマイクロ命令が順次読み出されて実行される（図3）。

図4にマイクロプログラム制御でマクロ命令が実行される様子の概念図を示す。イメージとしては，メモリ（マイクロROM），デコーダ，実行部（マイクロ操作生成）があることから，MPU内部に小型のMPUがある感じである。そもそも，歴史的に見れば，最初のMPUであるインテルの4004はプログラムを格納するROMの内容を交換することで，種々の電卓に対応しようとした。マイクロプログラムもマイクロROMの内容を交換することで種々の命令セットに対応できる。この奇妙な一致は偶然ではあるまい。

マイクロプログラム制御方式の利点は，制御部

図1 MPUの構成

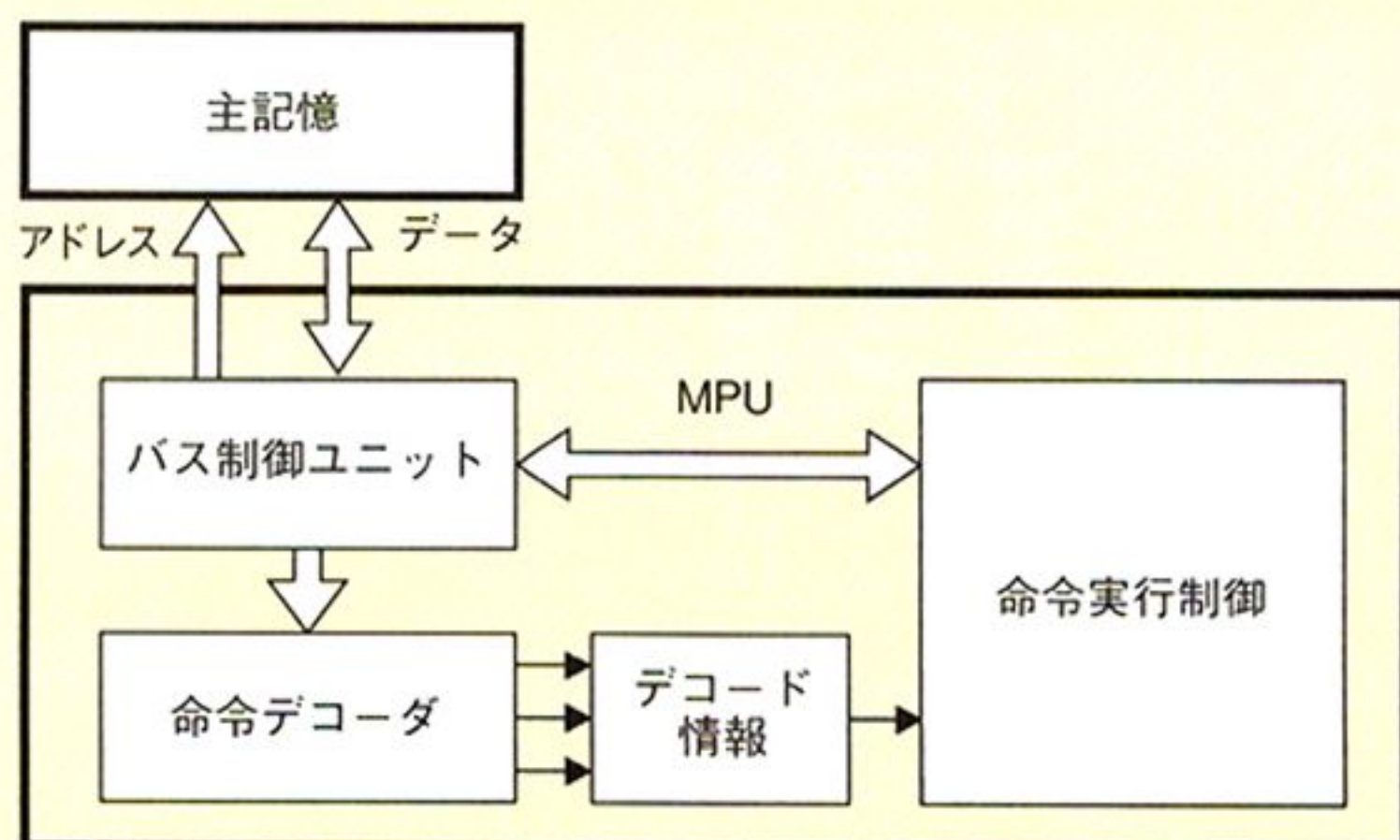
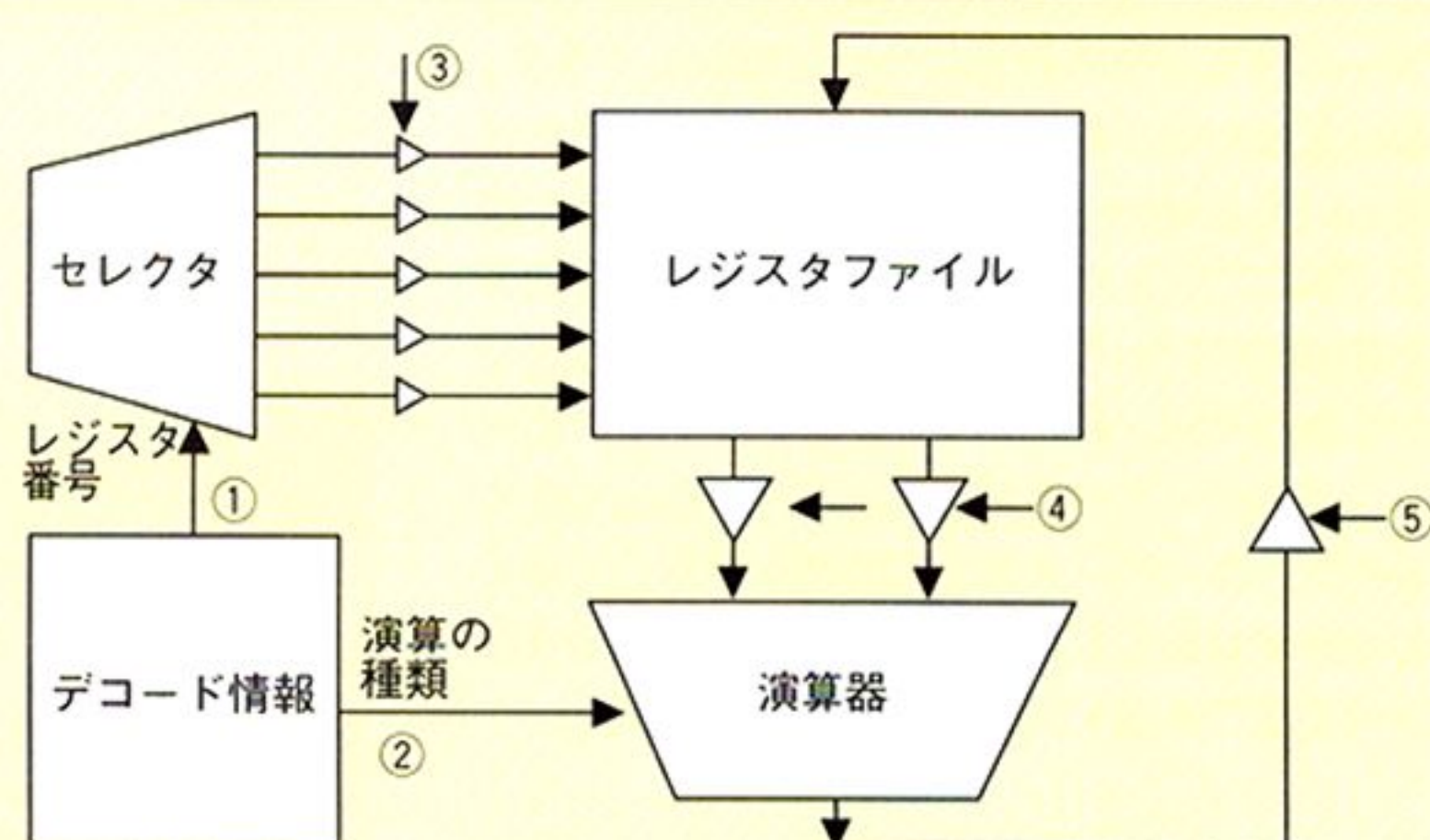



図2 マイクロ操作の例

の設計がすっきりしたものになることである。MPUの制御論理はマイクロプログラムで実現されるので，その論理設計はマイクロプログラムを作成することが中心となる。マイクロプログラムが完成する前に制御部のハードウェアを並行して(早期に)設計できる。これは命令セットアーキテクチャが定まっていない状態でもMPUの設計を始められることを意味する。あるいは，設計の後半で仕様変更が生じた場合も，マイクロプログラムを変更すれば，ハードウェアの変更なしで対応できる。その意味で命令の追加も容易である。

また，マイクロROMの内容を変更することでMPUの制御論理を容易に変更できる。極端な話，ひとつのハードウェアで複数の命令セットを実行すること(別のMPUのエミュレーション)が可能になる。マイクロプログラム次第でどのようなアーキテクチャのMPUにもなることができるのだ。

●マイクロ命令の詳細

以下では，マイクロROMに格納されるマイクロ命令について説明していく。

マイクロ命令は，通常，次のような情報から構成される。

(a) ステップの間に同時に実行されるマイクロ操作を指定する情報
(b) 次に実行するマイクロ命令を決定する順序制御に関する情報
(c) マイクロ命令で使用する定数

また，マイクロ命令はこれらの情報の表現形式によっていくつかの方式に分類することができる。大別すると，水平型(horizontal type, function field type)と垂直型(vertical type, machine code type)に分類できる。

(1) 水平型マイクロ命令

水平型のマイクロ命令は，マイクロ命令の各ビットが処理装置のゲートなどの制御点に1対1に対応している点に特徴がある。しかし，処理装置が複雑になると，当然，制御点も多くなり，マイクロ命令に必要なビット数が増加し，実用的でない。通常はエンコードすることでビット数を減らす方法を取る。具体的には，マイクロ操作信号を適当なグループに分けて，各グループごとにエンコードしてビット数を削減する。この場合，マイクロ命令を実行するためには各グループ(フィールドという)ごとにデコーダが必要である(図5)。マイクロプログラムの作成者にとっては，各サイクルごとのハードウェアの動き(各制御点の状態)を考慮してコーディングをしなければならないので，かなりの技術を要求される。

(2) 垂直型マイクロ命令

垂直型のマイクロ命令は，どちらかといえば，マクロ命令に近い。つまり，マイクロ操作のエンコードを複雑にすることにより，マイクロ操作に1対1に対応するという感覚がなくなってしまっている。図6に，その一例を示す。ここでFは操作コードであり，実行する動作を示し，O1，O2，……は操作オペランドフィールドで，演算回路の入力や結果の行き先を示す。マイクロプログラムの作成者にとっては，マクロ命令のプログラミングを行う感覚でコーディングできるので，親しみやすい。また，アルゴリズム記述向きの形式なので保守もしやすい。

垂直型のマイクロ命令は，ハードウェアとしては同時動作が可能なマイクロ操作であっても，エンコードの都合で，ひとつのマイクロ命令では同時に指定できないことがある。したがって，垂直型では水平型に比べてマイクロプログラムのステップ数が増加する傾向にある。高度なエンコードによってマイクロ命令のビット数を削減した半面，マイクロプログラムの容量増加につながる。

もっとも，現実のマイクロ命令は，水平型，垂直型の区別ができないものが多いのも事実である。両者の利点を折半した中間形式のマイクロ命令も多いらしい。この中間形式のものを対角型(diagonal type)ということがある。

●マイクロプログラムの挙動

図7にマクロ命令実行時のマイクロ命令の挙動を示す。マクロ命令のデコードが終了するとマイクロROMへのアクセスが開始され，次のクロッ

図3 マクロ命令とマイクロプログラム

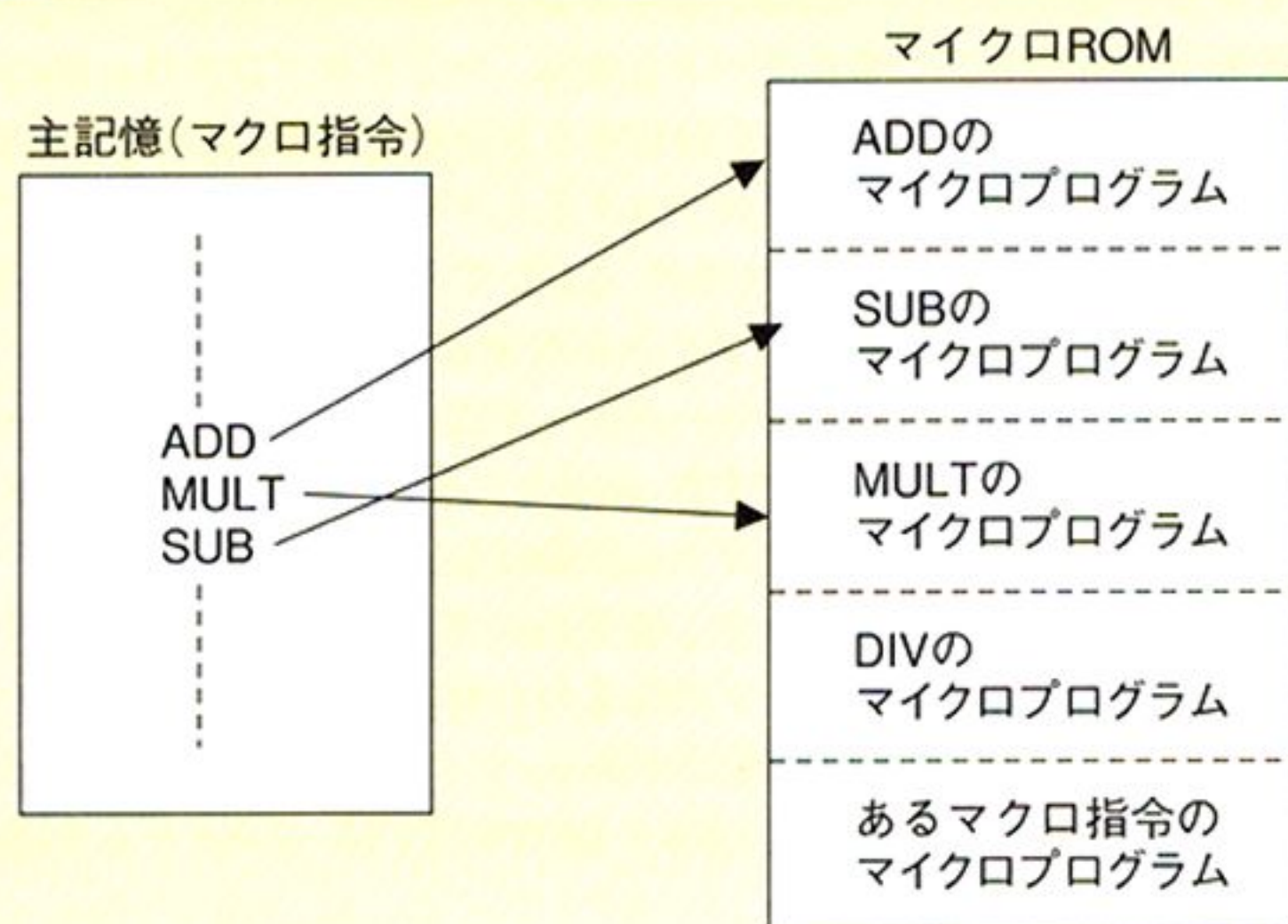



図4 マイクロプログラム制御の概念

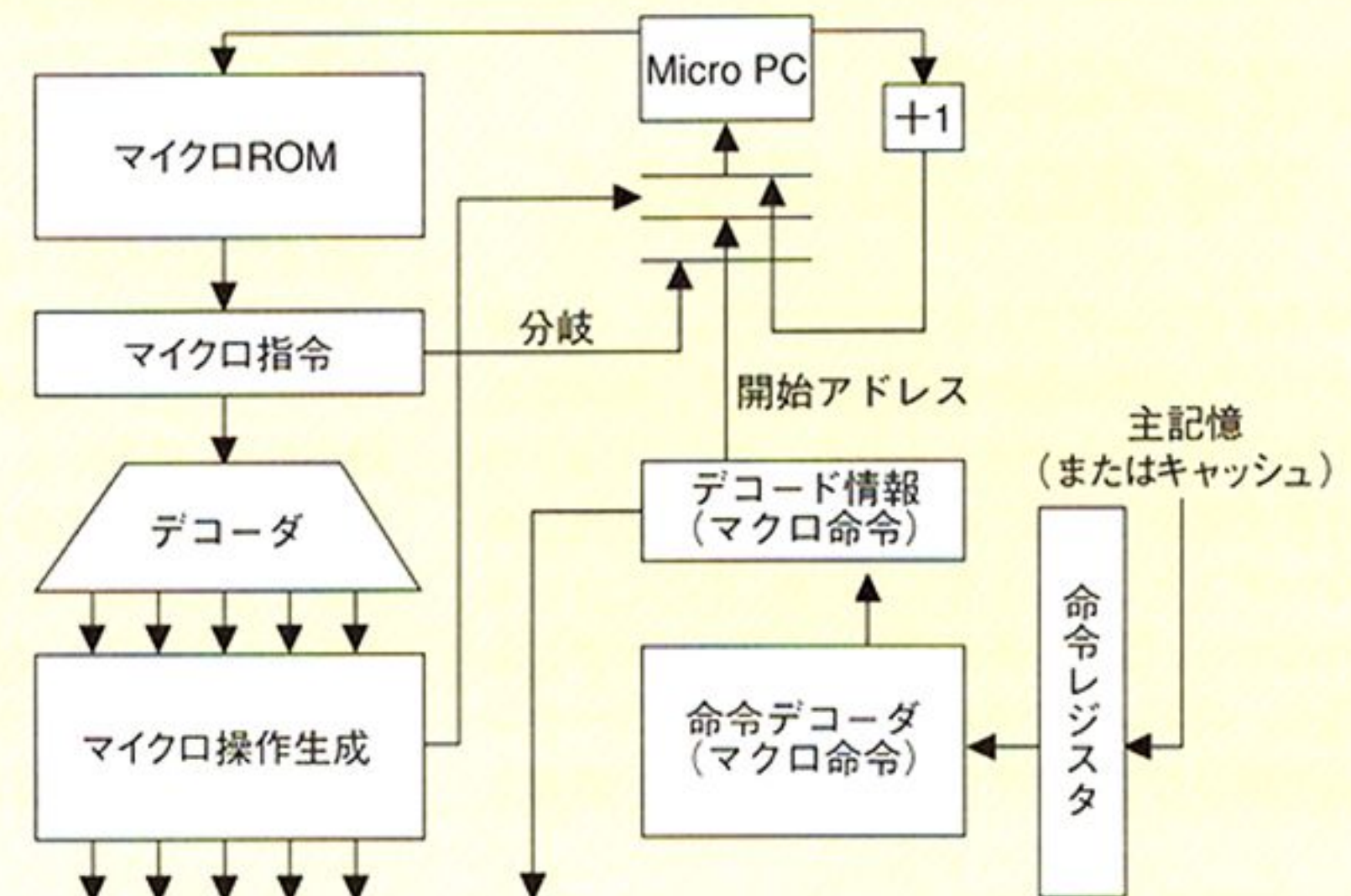



図5 水平型マイクロ命令

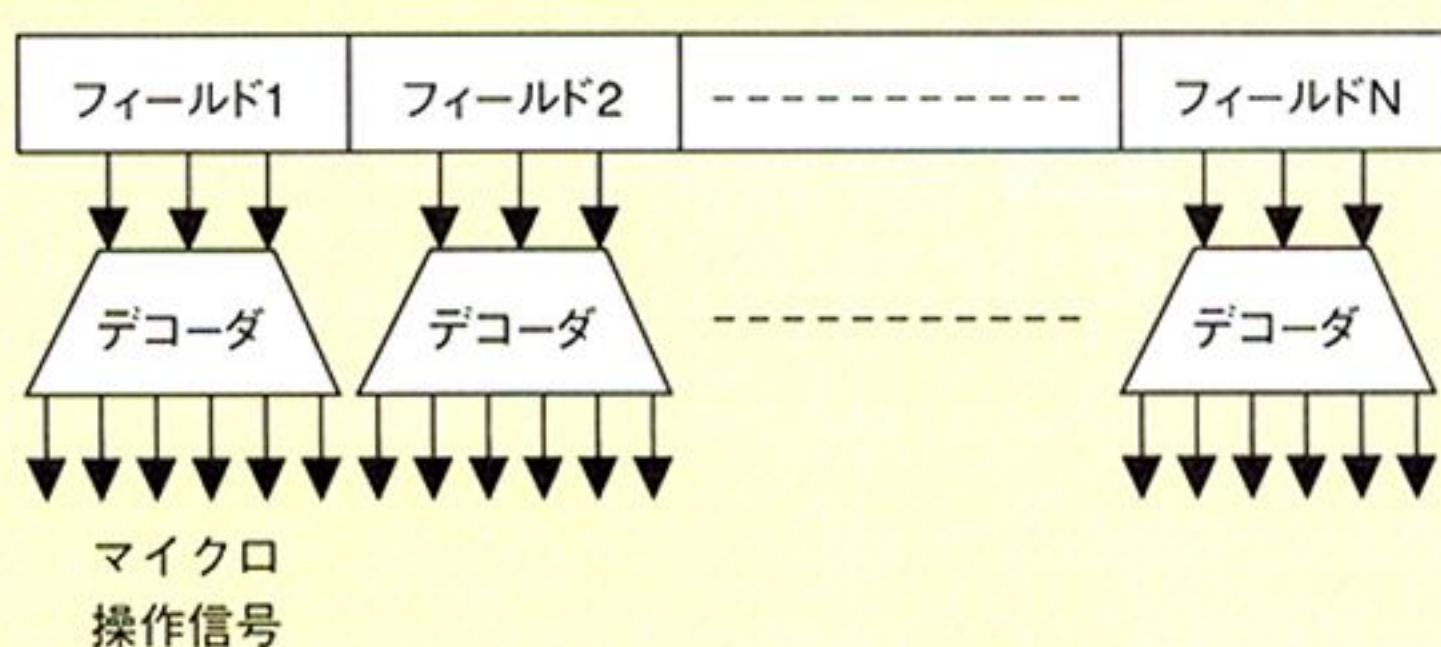
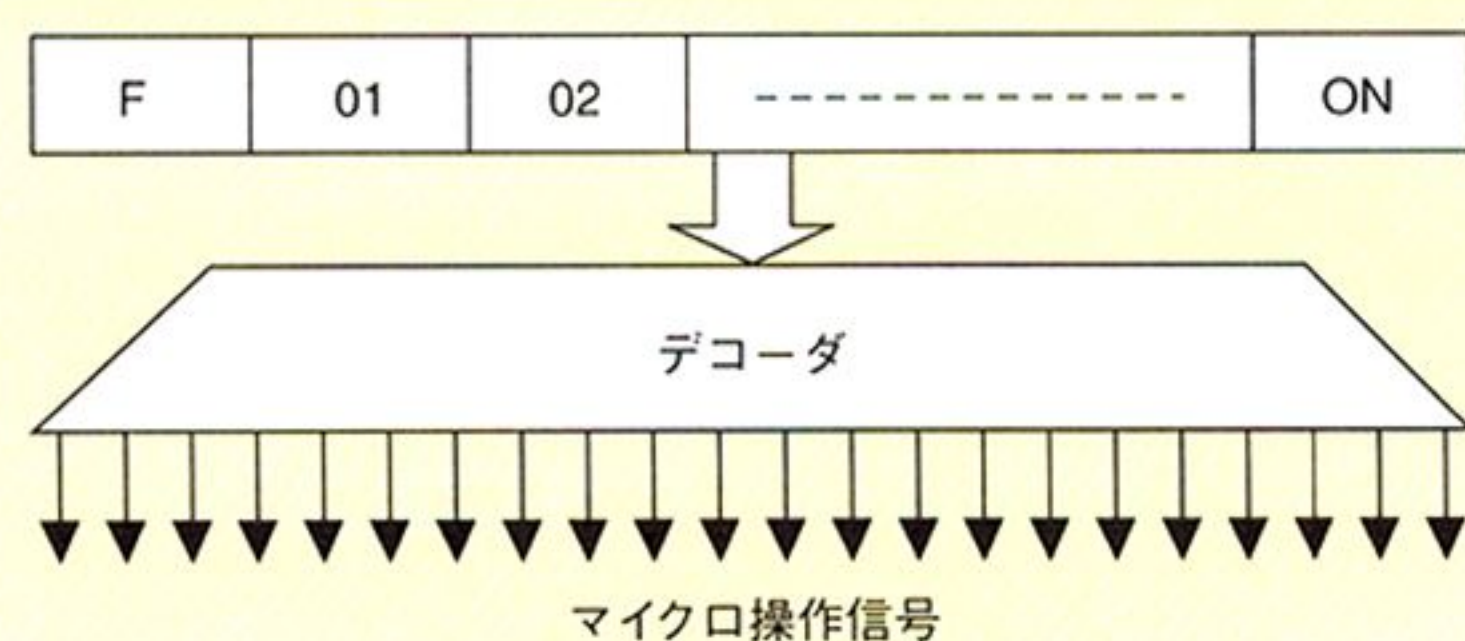



図6 垂直型マイクロ命令

クでマイクロ命令が出力される。図7ではマイクロ命令の読み出しと同時にデコードが行われ，すぐにマイクロ操作が実行される場合を示している。また，マイクロROMへはパイプライン式に毎クロックアクセスするものとし，マイクロ命令も毎クロック読み出せるものとしている。図7ではマクロ命令が4ステップのマイクロ命令で実現できる場合を示しているが，最初のROMの読み出しに1ステップかかるので，マクロ命令の実行には，実質5クロックかかる。

加減算のような単純な命令は，通常，1～2ステップのマイクロ命令で実行できるので，マイクロROMをアクセスするための1クロックは性能に影響を与える。これを省略する方法がある。もともと，加減算のような単純な命令は，たとえば，

(1) ソースオペランドを読み出し，演算器に演算の種類を設定する。
同時にオペランドを演算器に入力する。
(2) 演算結果をデスティネーションオペランドに書き込む。

というマイクロ命令で実行できる。これは，加算であろうが，減算であろうが，同じマイクロ命令で実現できる。つまり，マイクロ命令は固定しているので，単純な命令の場合は，最初の1ステップに関しては，マイクロROMを参照することなく，定数発生器でマイクロ命令を生成して実行してよい。この場合，定数発生に時間はかからないので，命令の実行を1クロック短縮できる。マイクロプログラムの2ステップ目以降は，1ステップ目からマイクロROMをアクセスしておけば，遅延なしでマイクロ命令を読み出せる（図8）。

●2レベルのマイクロプログラム

マイクロプログラムを2レベルにして，マイクロプログラムの融通性を増加させたり，変更修正を容易にすることが考えられる。マイクロROMの容量を減少させるためには，定型的な処理はサブルーチンとして1カ所にまとめ，そこをコールすればよい。特に，垂直型のマイクロプログラムの場合，エンコードが複雑なため，サブルーチンを水平型にしてマイクロ命令のデコードの負荷を軽減することもある。実際には，マイクロROMと別個に水平型マイクロ命令の格納されたROMを用意し，ある種（複雑な）のマイクロ命令の場合は，その付随するROMから水平型マイクロ命令を読み出して，デコード，実行する。

単純なマイクロ命令については通常のデコーダでデコードする。このような水平型のマイクロ命令は，マイクロ命令よりも細かい処理をするという意味で，ナノ命令と呼ばれる。複雑な命令や定型処理はマイクロ命令をデコードする前に，ナノROMのアドレス情報を生成するのである。図9にナノプログラムを使用する場合の構成図を示す。

単純な構成では，ひとつのマイクロ命令はひとつのナノ命令に対応する。この場合は実質，水平型のマイクロプログラムと大差ない。ただし，ナノROMの容量を削減するため，単純なマイクロ命令はそのまま，デコード，実行する。経験的に，（垂直型の）マイクロ命令のほうが，（水平型の）ナノ命令よりもビット長が短くなるからである。ナノプログラムをサブルーチン的に利用する場合は，ひとつのマイクロ命令に対して複数のナノ命令が対応する。つまり，マイクロ命令によってナノROMのアドレスが与えられると，そこから逐次的にナノ命令を読み出して，その処理の終了コマンドを実行するまで，実行を続ける。この方式は，アルゴリズムの記述にはマイクロ命令を用い，実際の（複雑な）ハードウェア制御にはナノ命令を用いることになるので，効率的にマイクロプログラミングが行える。

●マイクロプログラム制御方式の欠点？

これまで述べてきたように，マイクロプログラムはハードウェアのマイクロ操作信号を容易に生成でき，変更も容易であるし，保守もやりやすい。しかし，現在のMPUでマイクロプログラムが使用される場面は少ない（編注：最大手のインテル系列以外では）。その理由はなぜであろうか。それは，世の中のMPUの風潮がRISCになっているのと関係がある。そもそも，RISCは1クロックで実行できるような単純な命令しかサポートしない。その動作を実現するマイクロ操作信号も比較的単純である。こうなると，MPU内部に（マイクロ）ROMを持ち，（マイクロ命令の）デコーダを持つということは回路規模の増加になる。別にRISCをマイクロプログラムで制御しても構わないのだが，ハードウェア資源の無駄遣いである。つまり，マイクロプログラムが用いられない理由は，それに致命的な欠点があるわけではなく，必要がないからである。

x86アーキテクチャは，互換性のために，古きよき（?）CISCの命令をいまだにサポートしている。これら複雑な処理を行うCISC命令は，いまでもマイクロプログラムによって実現されている。よく，マイクロプログラムによる実行は遅いといわれるが，これはかなり誤解を招く表現である。マイクロプログラムで実行されるような複雑な命令はそもそも1クロックで処理することができないので，パイプライン処理に乱れが生じ，結果として命令のスループットが悪くなる。つまり，マイクロプログラムに原因があるのではなく，長くばらつきのある命令実行時間に問題があるのだ。

もっとも，マイクロプログラムを使用するとROMの読み出しに1クロック必要になり，命令の最低実行時間が2クロックになるので遅いという意見もあるだろう。基本命令に関しては，図8に示したような方法でROMの読み出し時間をなくすこともできるが，一般的には間違いではない。パイプライン制御を行う場合，スループットは最長のステージに律速されるからである。しかし，命令実行中は次の命令のデコード中なので，命令実行ステージが始まる前にマイクロROMのアクセスを開始して，最初の1クロックの遅延を稼ぐことも可能ではある。これは，マイクロROMのアドレスがデコード後に決定することを考えると，速度的に非常に厳しい。ただし，デコードと実行ステージの間にオペランドリードステージを新設すれば実現は可能である（パイプラインのステージが増加した分，分岐の性能が悪くなるが）。

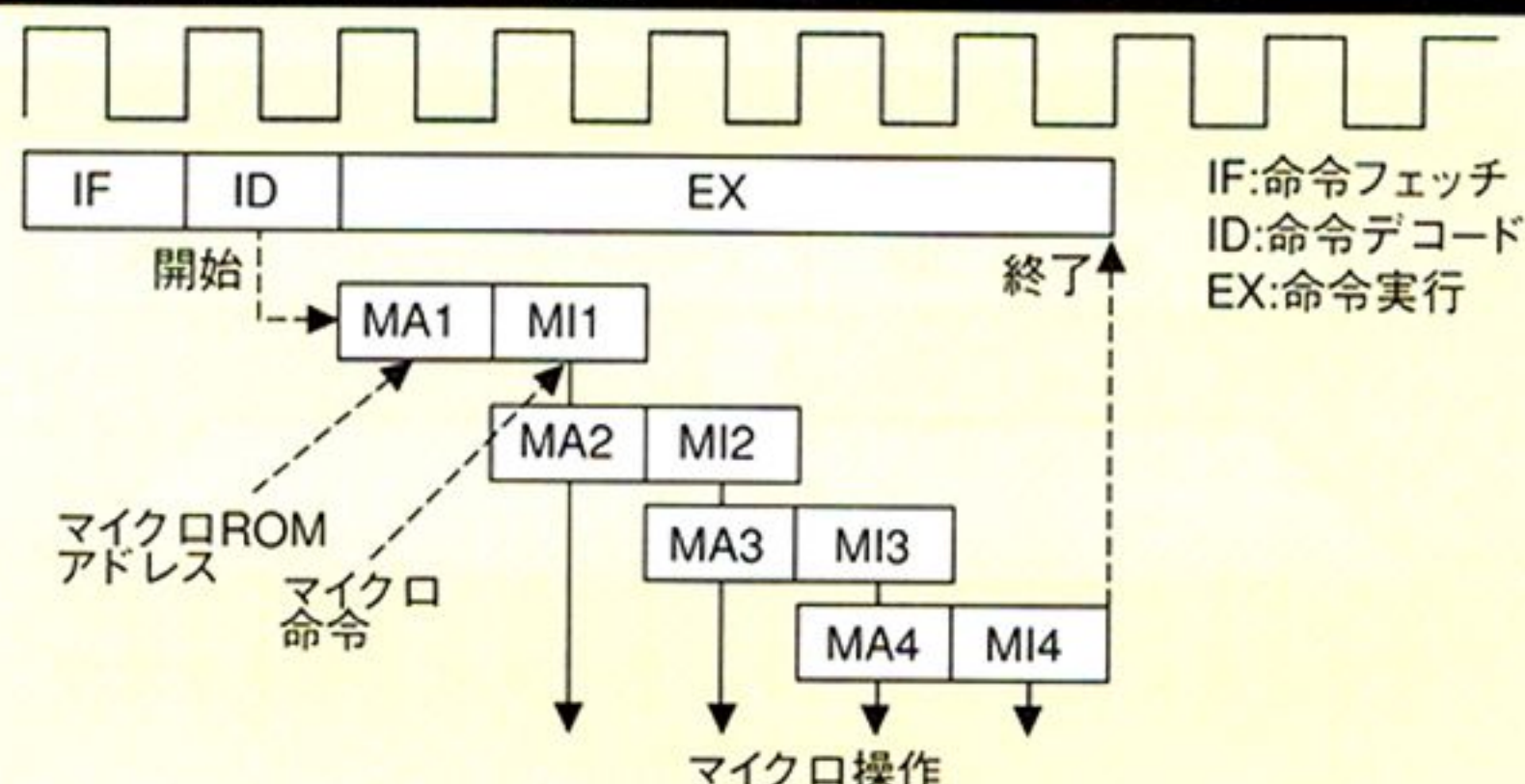
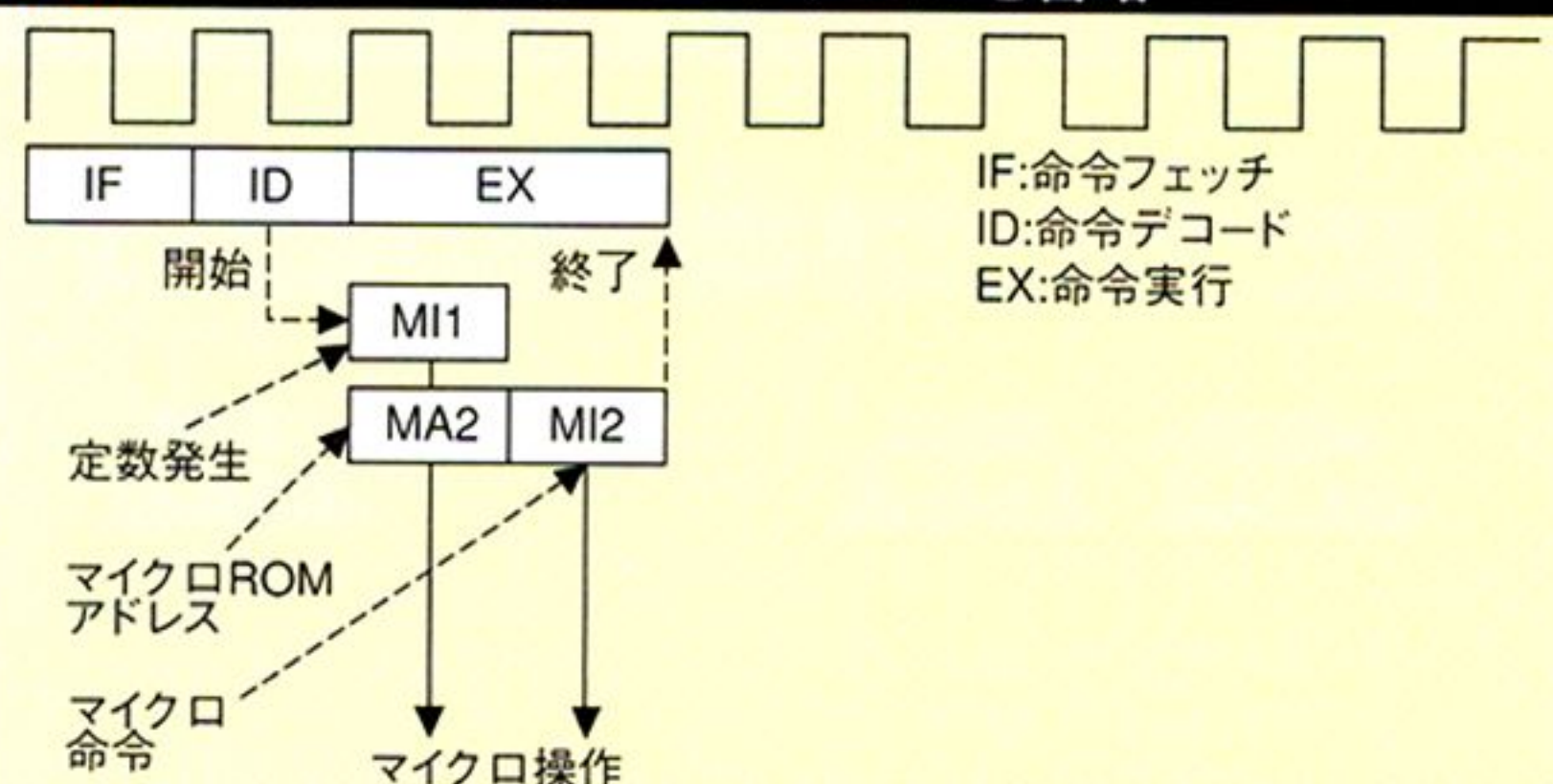
あるいはマクロコードをタグとし，マイクロ命令をデータとする，マイクロプログラム用のキャッシュを内蔵する方法もあるかもしれない。まあ，マイクロプログラミング制御を行うのは主としてCISCであり，CISCでそんなにスムーズに流れるパイプラインを有するMPUは稀なのだが。ここら辺のオーバヘッドはあまり考慮されてないような気がする。結論としては，うまく作れば，マイクロプログラム制御は全然遅くない。

もう少し歴史的に考察すれば，マイクロプログラミングの生まれた背景には，主記憶のアクセス時間の遅さがある。マイクロプログラミングが全盛だった1960～1970年代では，プログラムを格納す

図7　マイクロ命令の実行

IF ID EX
開始 終了
MA1 MI1
MA2 MI2
MA3 MI3
MA4 MI4
マイクロROMアドレス
マイクロ命令
マイクロ操作
IF:命令フェッチ
ID:命令デコード
EX:命令実行

図8　1ステップ目のROMアクセスを省略

IF ID EX
開始 終了
MI1
MA2 MI2
定数発生
マイクロROMアドレス
マイクロ命令
マイクロ操作
IF:命令フェッチ
ID:命令デコード
EX:命令実行

る主記憶に磁気コアメモリが用いられていた。これは，半導体メモリ（ROM）と比べると，10倍程度アクセス時間が遅い。そこで，主記憶からフェッチするマクロ命令のコードサイズを小容量，多機能にし，それをアクセス時間の速いROMに格納されたマイクロプログラムで高速に実行することが考えられた。MPUの命令セットはマイクロプログラムで実行されることを前提として設計されている。これは，まさにCISCの考え方である。しかし，1970年代の後半からは，キャッシュメモリというアクセス時間の速いメモリを主記憶として利用することが可能になり，マクロ命令のフェッチとマイクロ命令のフェッチに時間差がなくなった。こうなると，上述したように，わざわざマイクロプログラムを用いる利点がなくなる。将来的には，キャッシュメモリに比べて非常にアクセス時間の短いROMが開発されるか，キャッシュメモリ自体が有効でなくなるような技術的革新が起きない限り，マイクロプログラミングの復権はなさそうである。

ところで，MPUが実行するマクロ命令がマイクロ命令そのものだったらどうだろう。その場合は，プログラムの実行を最高速で，しかも並列に，実行することが可能になる。事実，DSP（Digital Signal Processor）でのプログラミングはマイクロプログラムの香りを残している。プログラムはDSPの内部ユニットと密接に関連しており，プログラマはその内部状態を考慮して最適な動作になるように調節する必要がある。できるだけ多くのユニットを並列に動作させるようにするのがキーポイントである。そのため，（アセンブラによる）プログラムは非常に難しいが，これによって得られる性能は非常に高速である。世間的には「DSP＝高速プロセッサ」というのが常識であるが，これはプログラムがハードウェアをもっとも高い効率で動作させるためである。以下に述べるVLIWに関しても，そのようなイメージがついてまわる。読者の方々はどのように思われるであろうか。

● VLIWとは

VLIW（Very Long Instruction Word）とは，その名称のとおり「非常に長い命令語」を意味する。一般に，命令は128ビット程度の固定長で，MPUが持つ機能ユニットと1対1に対応する「スロット」という領域から構成される。スロットには対応するユニットを制御する命令が埋め込まれ，VLIWの1命令が実行されると，各スロットの命令が同時実行される。つまり，1ステップで複数の命令が実行される。図10にVLIWの概念図を示す。VLIWでは，スーパースカラとは異なり，ハードウェアが同時実行できる命令を自動判別するわけではない（命令内で明示的に指定されている）ので，命令発行ユニットを簡略化できる。

一方，それぞれの命令の各スロットが最大限に機能するように割り当てる必要があり，この非常に技術を要する負荷をコンパイラに課すことになる。コンパイラは与えられた命令列から同時実行可能な命令の組を探し出し，VLIW命令の各スロットに割り当てる。スロットに入るべき命令がない場合はNOP（No Operation）を入れる。これを命令スケジューリングという（図11）。

スーパースカラとVLIWを比較すると，MPU内部に同時実行を行う仕組みを実装する必要がなくハードウェアを簡略化できるという利点がある半面，コンパイラが頑張っても，常に最高の個数（スロットの数）の命令を同時実行できるとは限らず，NOPが余分に発生する分だけプログラムのコード効率が悪くなる（コードサイズが増加する）という欠点がある。

VLIWの発想は，Control DataのCDC6600やIBM 360/91という最初のスーパーコンピュータで採用されていたマイクロコードの並列実行方式に由来する。それらのコンピュータが活躍した1960～1970年代では，アレイプロセッサや専用のシグナルプロセッサがROMに格納されたVLIWによく似た語長の長い命令を用い，高速フーリエ変換などのアルゴリズムを計算していた。

真のVLIWマシンは1980年代初期に3つの会社（Multiflow, Culler, Cydrome）から発表されたミニスーパーコンピュータだった。それらは商業的には成功しなかったが，これらのコンピュータに適用されたコンパイラ技術は無駄にはならなかった。その後，HP（Hewlett Packard）はMultiflowを買収し，現在のHPのVLIWのコンパイラ開発はMultiflow出身のJosh FisherとCydrome出身のBob Rauを中心に行われているという。トレーススケジューリングとソフトウェアパイプライニングは，それぞれ，FisherとRauが先駆者であるが，現在のVLIWコンパイラ技術の中心的役割を果たしている。なお，VLIW用に開発されたコンパイラの並列化技術の多くはスーパースカラ用のコンパイラに採用されて成功を収めているのは痛烈な皮肉である。

ところで，VLIWの嚆矢となったMultiflow-7/300は2つの整数ALU，2つの浮動小数点ALU，分岐ユニットを有していた（これらは複数のチップで構成されていた）。その256ビットの命令語は7つの32ビット長のオペレーションコードを含んでいた。各整数ユニットは130nsごとに2つのオペレーションを実行できた（都合4命令）ので，約30.8MIPSという性能となる。当時としてはかなり高性能な部類だ。また7/300を複数組み合わせて，より高性能な512ビットや1024ビット幅のマシンを構成することもできた。

一方，CydromeのコンピュータであるCydra-5は，各命令を6つの40ビットの操作として順次実行する特殊モードを備えてはいたが，256ビットの命令語を使用していた。そのため，そのコンパイラは並列コードと従来どおりの逐次的コードをミックスしたコードを生成したという。Cydra-5は，Multiflowとは異なり，プレディケーション，ソフトウェアパイプラインといったVLIWコンパイラの基本的な並列化技術をすでに採用していたといわれている。

これら両方のVLIWマシンは複数のチップで構成されていたが，最初の1チップのVLIWは1989年に発表されたIntelのi860であるといわれている。2つの32ビット命令（整数と浮動小数点数）を64ビット命令とみなして一度に命令キャッシュから取り込んで同時に実行するデュアル命令モードを備える。64ビット長なのでLIW（Long Instruction Word）と呼ぶのが正しいかもしれない。i860は，世間にはDSPとして受け入れられ，もっぱらグラフィックアクセラレータとして利用された（最近はRAIDコントローラ用途のほうがメジャーかも）。なお，i860では，命令を正確に実行する責任はハードウェアよりもむしろコンパイラに任せていた。結果としてi860は成功しなかったが，すべ

図9 ナノプログラムの概念

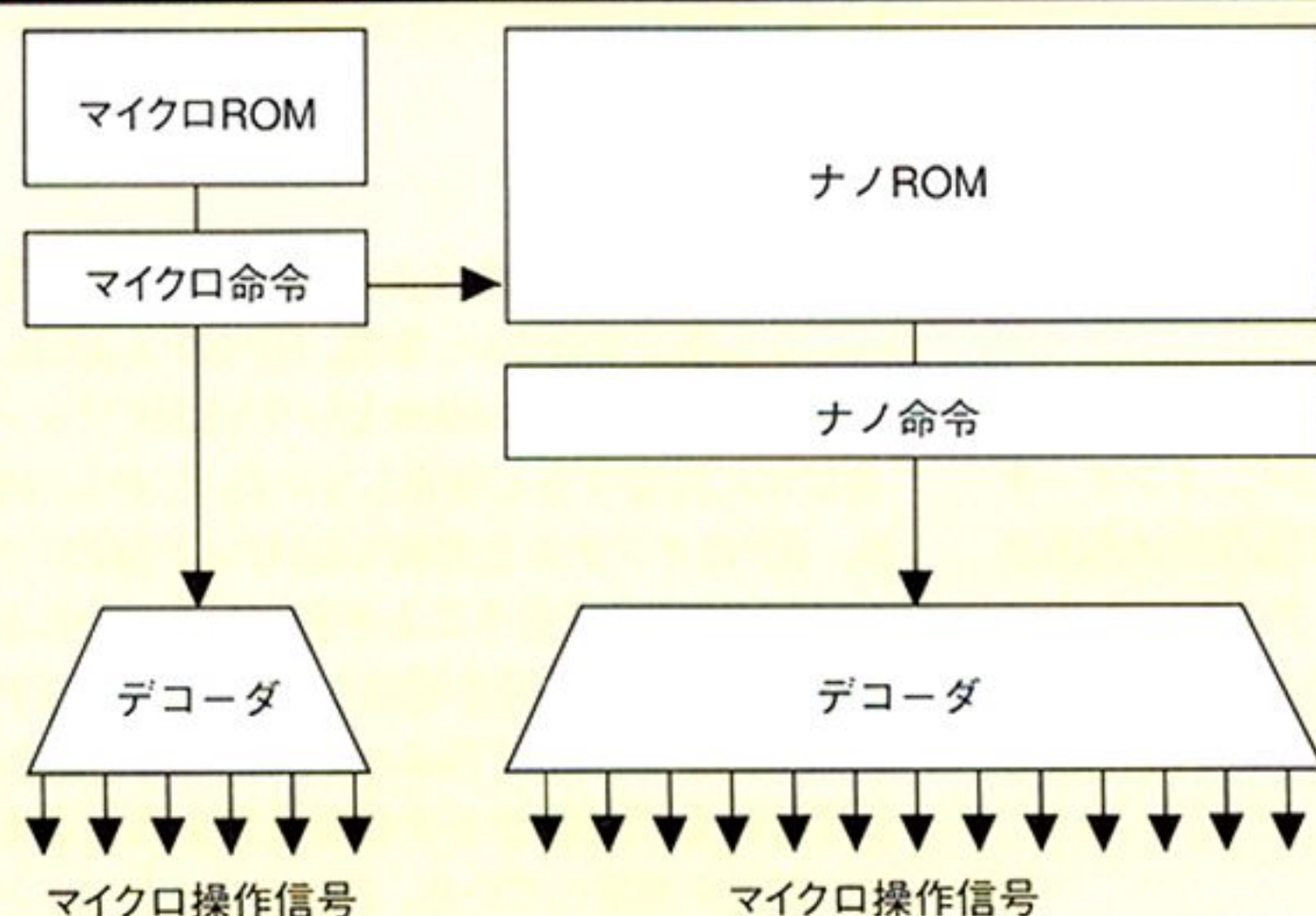



図10 VLIWの概念

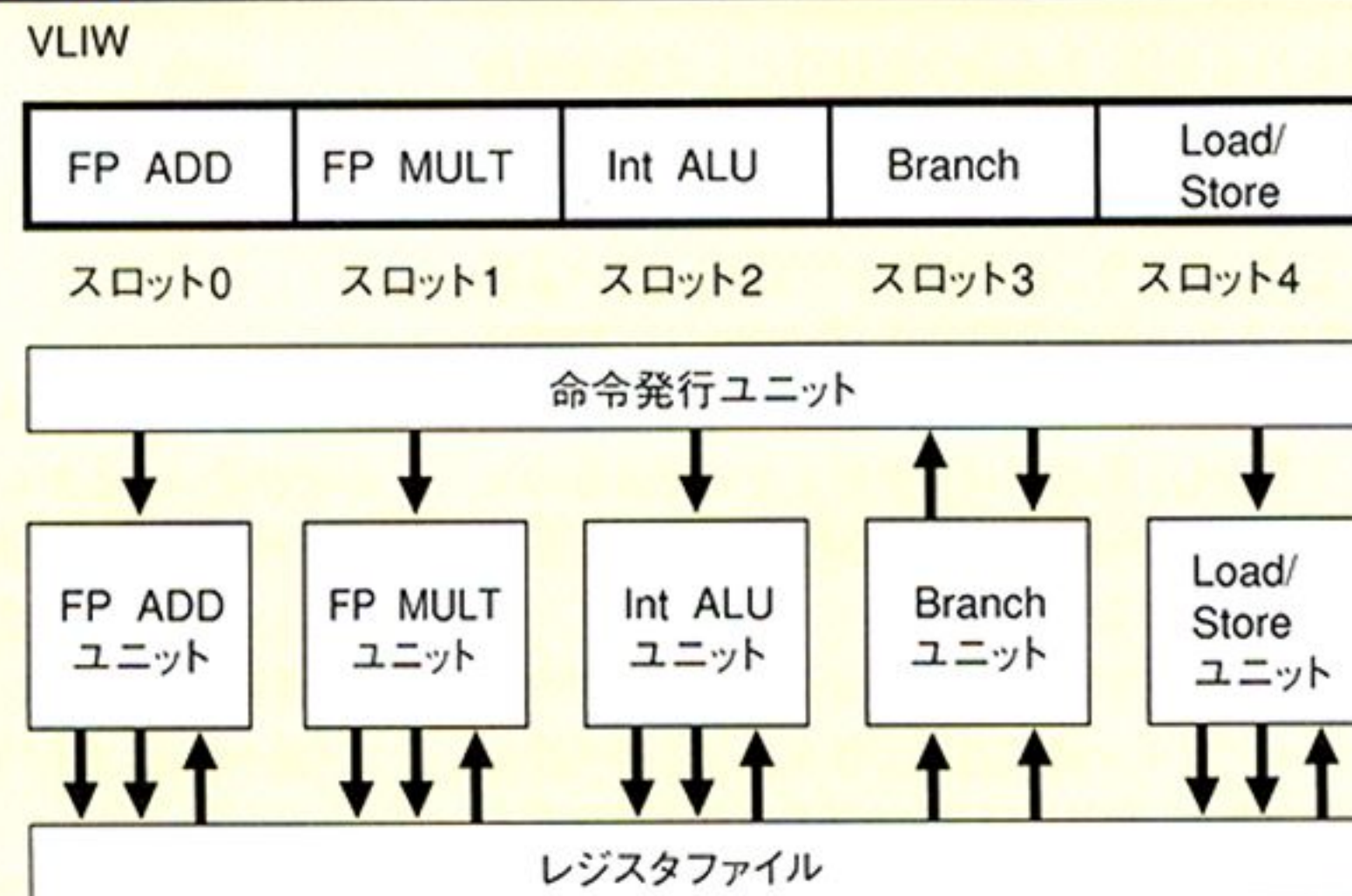



図11　コンパイラによる命令スケジューリング

命令例
命令0
命令1
命令2
命令3
⋮
命令N

スロット0	スロット1	スロット2	スロット3	
-----	-----	-----	-----	6
-----	-----	-----	-----	5
-----	-----	-----	-----	4
命令7	NOP	命令8	命令9	3
NOP	命令6	NOP	NOP	2
命令3	NOP	命令4	命令5	1
NOP	命令1	命令2	NOP	0

図13　Itaniumの命令フォーマット(バンドル)

図12　x86とIA-64の比較

	x86	IA-64
命令形式	複雑で可変長の命令を一度に1命令処理する	単純で固定長の命令が3つをひとつのグループにバンドルして同時に処理する
実行順序	命令列の並べ替えと最適化を実行時に行う	命令列の並べ替えと最適化をコンパイル時に行う
分岐予測	予測した分岐先を投機的に実行する	分岐先と分岐元の両方を投機的に実行し，不要な側を無視する
メモリ参照	必要になったときにデータをメモリからロードする。キャッシュを最初に参照	必要になる前に投機的にデータをロードする。同じくキャッシュを最初に参照

てソフト任せというハードウェアの自由度の低さが一因だったのではなかろうか。結局，コンパイラがうまく開発できなかったのだろう。

ただ，当時のコンパイラ技術では，プログラム中で並行に実行できるのは2命令程度という報告があった(5命令という報告もあったようだが)。また，先進的な並列化技術もコンピュータの非力さゆえにコンパイル時間が現実的でなく，VLIWの登場は時期尚早という感もあった。

●トレーススケジューリングとソフトウェアパイプライニング

VLIWのコンパイラの基本技術である，トレーススケジューリングとソフトウェアパイプライニングについて簡単に説明しておこう。

コンパイラは，ある命令を起点として命令列を探索し，それが分岐に突き当たり，プログラムの流れが変わるまでを基本ブロックとして最適化処理を行う。つまり，命令がループになっている場合，異なるループ処理間では，基本的には，同時に最適化を行えない。しかし，(おそらくはひとつのループを含む)基本ブロックをまたいで命令のスケジューリング(並べ替え)を行う手法がトレーススケジューリングである。

ソフトウェアパイプライニングとはループの内部をパイプライン的に処理できるように並べ替える手法である。それには，ループアンローリングという手法が基本となる。これはループをアンロール(部分的に展開)するものである。

たとえば，

```
for (i=1;i<=N;i++) {
  命令1
  命令2
  命令3
  }
```

というループを考える。これを，アウトオブオーダーなスーパースカラで実行する場合，命令1，命令2，命令3の3命令では依存性が高くてうまく並列実行できないことがある。その場合，たとえば，2ループをひとつの単位として，

```
for (i=1;i<= (N/2) ;i+=2) {
  命令1
  命令2
  命令3
  命令1
  命令2
  命令3
  }
```

とアンロールしてみれば，並列実行できる組み合わせが見つかるかもしれない。しかし，インオーダーなパイプラインでは相変わらず依存性は解消されていない。そこで，ソフトウェア(コンパイラ)で，並列実行できるように命令のスケジューリング(並べ替え)を行うことが考えられる。たとえば，

```
for (i=1;i<= (N/2) ;i+=2) {
  (命令1 ; 命令1)
  (命令2 ; 命令2)
  (命令3 ; 命令3)
  }
```

のようにスケジューリングすれば(；)内の組が並列に実行できるとすれば，処理時間が半分になる。このようなアンローリングしたループ内の命令の並べ替えをソフトウェアパイプライニングという。パイプラインというイメージからは，

```
for (i=1;i<= (N/2) ;i+=2) {
  (命令1 ;       )
  (命令2 ; 命令1)
  (命令3 ; 命令2)
  (       ; 命令3)
  }
```

のように，命令がオーバーラップする感覚で，並べ替えられることを想定しているのかもしれない。

●VLIWの実際

VLIWの実現性を疑問視する声が多いのは事実であるが，最近では珍しく成功した(といっていいのかな)2つのMPUがある。インテルのItanium(コードネームはMerced)とTransmetaのCrusoeである。どちらもx86命令を実行する方式としてVLIWアーキテクチャを採用している点が興味深い。これらの特徴を見ていこう。

[1] Itanium

(1) 開発背景

VLIWはHPの技術によって支えられているといっても過言ではない。事実，HPはPA-RSICの最新機種として，PA-9000というVLIWマシンを1998年に発売すると発表していた。しかし，1994年，HPはインテルと共同で64ビットMPUであるIA-64の開発を行うことを表明した。それと同時にPA-9000の開発を凍結した。IA-64はEPIC(Explicity Parallel Instruction Computing)と呼ばれるVLIWライクな命令を主体とする新しい概念を採用している。EPICは命令セットア

ーキテクチャと同時に高性能化などのインプリメンテーションを同時に定義する。

x86とIA-64の主な特徴の比較を図12に示す。64ビットRISCとしては後発に当たるため，新しい技術を提供することが必要であり，それにより明るい未来が約束されているように思えた。しかし，共同開発とは名ばかりで，実質的なIA-64の開発はインテルによって行われた。当初の計画とは異なり，PA-RISCとの互換性はなくなり，IA-64とx86のみのサポートになった（妥協案としてダイナミックトランスレーションというエミュレーション技術が提案されてはいるが）時点で，HPは共同開発から手を引いた感がある。

実際，HPは1999年6月からPA-RISCの最新機種であるPA-8500の出荷を開始した。同時に，PA-8600/8700/8800/8900というロードマップを発表してPA-RISCの存続をアピールした。PA-8500を使用したサーバであるHP9000はIA-64にアップグレード可能となってはいるが，それはIA-64の最初のItanium（Merced）ではなく，次機種のMckinley（コードネーム）であるとしている。HPのいい分は「Itaniumの性能は期待はずれ」ということらしい。性能問題が尾を引いたのか，Itaniumの開発は遅れに遅れ，2000年に発売に漕ぎつけたものの，その年の7月にはさらなる改版が必要として，2001年の前半まで発売を延期した。発表された動作周波数も800MHzと予想を遥かに下回っている。HPの公式見解はいまだにIA-64に期待するというものであるが，Itaniumの失態が繰り返されるようではHPの離脱もありえる。SGIなどもItanium搭載のLinuxマシンを発表しているがどうなるのだろう。

(2) 命令フォーマット

Itaniumの命令フォーマットを図13に示す。41ビットの3オペランド命令が3個と5ビットのテンプレートを組み合わせた128ビット長の命令である。これはバンドル（抱き合わせ）と呼ばれる。

各スロットに格納される命令のプレディケートとは「述語」という意味で，命令の実行条件を示す。分岐属性とも訳される。Itaniumは64本の1ビット長のプレディケートレジスタを備え，各レジスタは「真」または「偽」の情報を保持する。命令中のプレディケート領域はプレディケートレジスタの番号を表し，そのレジスタが「真」ならば命令を実行する。これはRISCで採用されている条件MOVEの発展形で，条件分岐での場合分けを（性能が低下する）分岐命令を使わずに表現できる。また，各命令は3オペランド命令である。

レジスタは128本用意されているが，常時使用できるレジスタはR0～R31の32本のみである。R32～R127はallocという命令でスタックフレーム領域として定義すると使用可能になる。サブルーチンコール先で確保されたスタックフレームはサブルーチンからリターンすると自動的に消滅する。このようなレジスタ構造をIA-64ではスタックレジスタと呼んでいる。

バンドルに含まれるテンプレートは3つのスロットの命令の種類を指定する領域である。これは内蔵する演算器と密接に関連し，I（整数），M（メモリ），F（浮動小数点），B（分岐）の中から指定できる。テンプレートは同時に並列実行できる命令の区切りも指定する。ハードウェア資源（演算器の数）が十分にあれば，基本的に，すべての命令を並列実行可能であるが，実装する演算器の数に依存して区切りが決定される。これは5ビットの領域なので32種類の組み合わせを指定可能であるが，Itaniumでは図14に示す24種類が定義されている。

なお，バンドルは1対1にPA-RISCの命令に変換可能というが，実際はどうなのだろう。

(3) 内部構造

図15にItaniumのブロック図を示す。演算器の構成は，分岐ユニット3個，整数/メモリユニット4個，浮動小数点ユニット2個である。2つのバンドルが同時に処理されるので6命令同時発行が可能である。また，L1，L2キャッシュを内蔵し，外部にはL3キャッシュが接続される。

Itaniumがもっとも効率よく命令の並列実行を行うためには，コンパイラによるヒント（制御情報）を命令に反映させることが肝要である。VLIWではコンパイラによるいろいろな並列化技術が研究されているが，それらをいかに適用できるかでMPUの性能が決まるといっても過言ではない。図16に各ハードウェアブロックに対してどのようなコンパイラ技術が必要になるか，その一例を示す。

(4) パイプライン

図17にItaniumのパイプラインを示す。10ステージからなるインオーダーなパイプラインである。パイプラインは，フロントエンド，命令供給，オペランド供給，実行の段階に大別される。

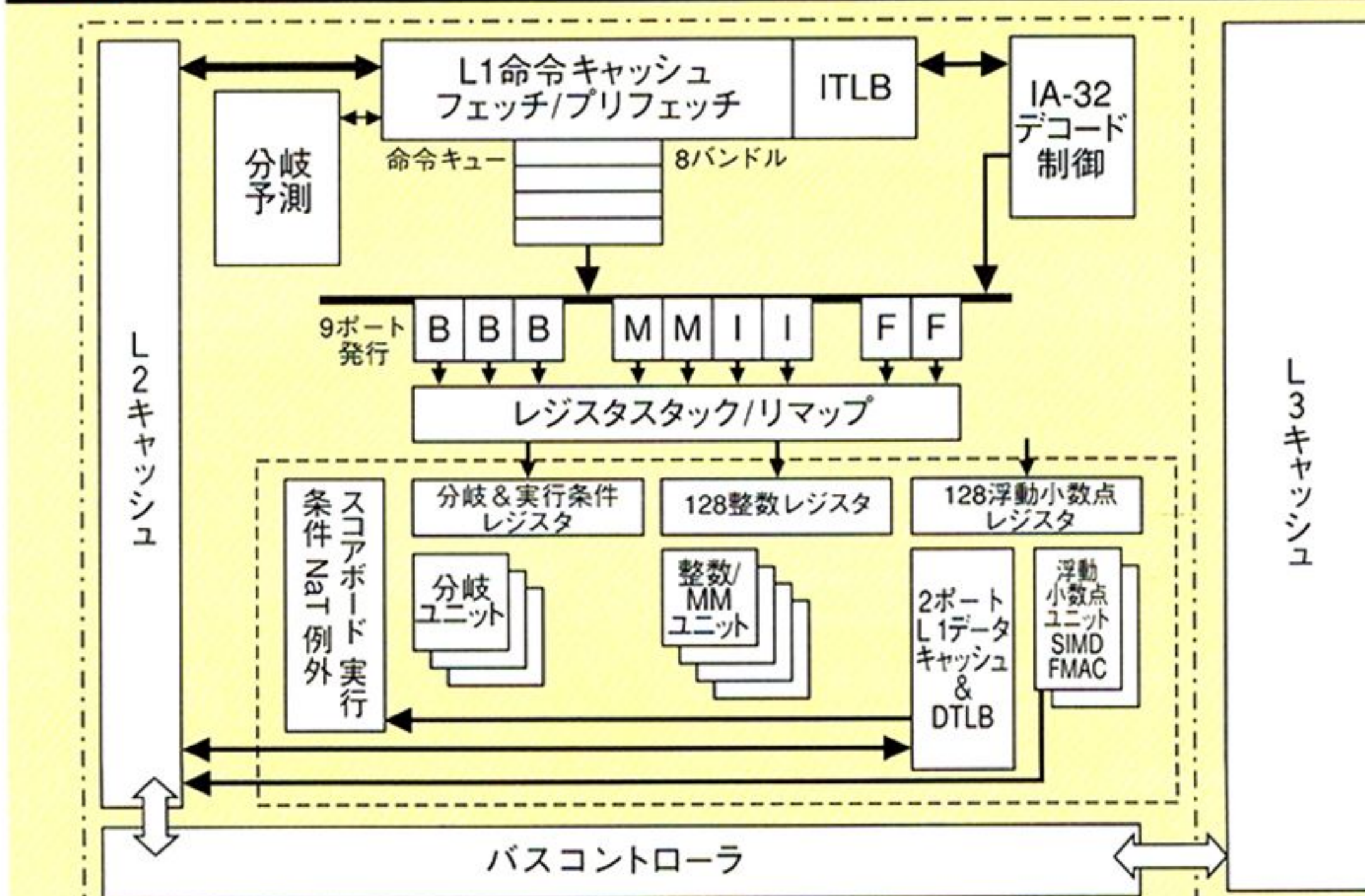
フロントエンド（図18）の段階では，1サイクルに最大6命令（2バンドル）をプリフェッチまたはフェッチし，8エントリのデカップリングバッファ（命令キュー）に格納する。分岐予測もここで行う。

命令供給の段階（図19）では，最大6命令を9つのポートに振り分ける。これはDispersal Network（拡散ネットワーク，とでも訳すか）によって行われる。将来的には各スロットがすべてのポートに振り分け可能になる予定であるが，Itaniumではスロットとポートの対応に制限がある。レジスタのリネームやスタック操作（動的な

図14 バンドルの組み合わせ

テンプレート	スロット0	スロット1	スロット2
00	M	I	I
01	M	I	I
02	M	I	I
03	M	I	I
04	M	I (L+X)	
05	M	I (L+X)	
08	M	M	I
09	M	M	I
0A	M	M	I
0B	M	M	I
0C	M	F	I
0D	M	F	I
0E	M	M	F
0F	M	M	F
10	M	I	B
11	M	I	B
12	M	B	B
13	M	B	B
16	B	B	B
17	B	B	B
18	M	M	B
19	M	M	B
1C	M	F	B
1D	M	F	B

▌並列実行できる命令の区切り

図15 Itaniumのブロック図

レジスタ割り当て＝Spill/Fill）もここで行う。

オペランド供給の段階（図20左）では，レジスタリードとバイパス（フォワーディング）処理を行う。そのほか，オペランドの依存性のチェック，分岐属性の依存性チェックを行う。

実行の段階（図20右）では，4つの1サイクルレイテンシのALU，2つのロード/ストアユニットで命令を実行する。また，投機ロードの制御，分岐属性の処理，例外検出，リタイア処理（図21）を行う。

(5)感想

Itaniumのパイプラインは単純なインオーダーな10ステージ構成である。その性能のキーポイントは2つのバンドル（6命令）をいかに並列実行できるかにかかっている。しかし，それはコンパイラでのスケジューリングに大きく左右される。そもそも，通常のアプリケーションプログラムで6つの命令を並列に実行できるほど命令間の依存度の低い場合がありうるのだろうか。トレーススケジューリングやソフトウェアパイプライニングなどの技法でそれなりに並列度を上げることができたとしても，実際のところどうだろう。まあ，将来的に，コンパイラの技術が進歩すれば進歩するほど性能が上がる構成になっていると考えれば，先見の明ということもできるのだが。

[2]Crusoe

(1)開発背景

1999年の半ば頃から，Linuxの作者として有名なLinus B. Torvaldsが開発に係わっているMPUが密かに開発中という噂が流れていた。そのMPUはいままでになく画期的なものという触れ込みだったので業界の注目を集めていた。そして，2000年1月，ついにTransmetaからx86互換MPUであるCrusoeの発表があった。従来，インテルやAMDは，x86アーキテクチャの命令を高速に実行するために，x86の命令をRISCライクな命令に変換し，それをスーパースカラで並列実行するという方式を採用していた。この方式はパイプラインの複雑な制御が必要で，回路規模がかなり大きくなっている。当然，消費電力も大きくなる。現在のインテルやAMD製のモバイル向けMPUは，非常に優秀な省電力技術と電力消費を抑える優れた製造技術で作られているが，それでもモバイル向けとするには厳しいのが実情である。それに対し，TransmetaはVLIWを採用し，回路構成を単純にする試みを行った。

図22に示すように，Crusoeでは従来はすべてハードウェアで行っていた処理をハードウェアとソフトウェアの組み合わせで実現する。資料によれば，0.22μmプロセスで製造されるTM3120のチップ面積が77mm^2，0.18μmプロセスで製造されるTM5400が73mm^{2}である。0.18μmプロセスで製造されるPentium Ⅲのチップ面積が106mm^2，Pentium4のチップ面積が217mm^{2}であることを考えると，その回路規模の小ささが実感できる。

なお，VLIWとしては128ビット長の命令を採用し，4つの操作を1命令に収める。VLIWの命令をx86命令に見せかける手法は，コードモーフィングソフトウェア（Code Morphing Software：CMS）と呼ばれるソフトウェアでx86命令をVLIWのネイティブ命令へとコンパイルするというものである。変換された命令はメモリ上に置かれた命令キャッシュ（まともな性能を得るためには16Mバイト程度の容量が必要）に格納され，高速に実行される。Crusoeの方式をTransmetaの言葉で表現すれば「プロセッサコアに組み込まれていたマイクロコードが外部ソフトウェアとして実装された」ということである。つまり，マイクロ命令（VLIWに変換されたx86命令）を直接実行するマイクロプログラム制御方式のMPUという考えである。

気になるCrusoeの性能は，Transmetaの発表では，700MHz動作のTM5400が500MHz動作のPentium Ⅲ相当ということで，それほど高性能ではない。実際の製品のベンチマークでも，600MHz動作のTM5600の整数性能が400～500MHzのPentium Ⅲ相当であることが実証されている。浮動小数点演算の性能は思ったよりも悪いようである。それでも，「LongRun」と呼ばれる電源管理機能によって，従来のモバイル向けx86互換チップと比べると，1W以下という超低消費電力を実現しているため，ノートPCでの採用が相次いでいる。冷却用のファンが不要なことも採用の一因であろう。

(2)命令フォーマット

CrusoeのVLIWエンジンは2つの整数ユニット，ひとつの浮動小数点ユニット，ひとつのメモリ（ロード/ストア）ユニット，ひとつの分岐ユニットから構成される。CrusoeのVLIW命令はモ

図16　コンパイラ技術とハードウェア

コンパイラで対応する技術
分岐のヒント　明示的並列性　レジスタスタック&ローテーション　プレディケート（分岐属性）　データ&制御投機的処理　メモリアクセスのヒント
フェッチ：命令キャッシュ&分岐予測
発行：高速，単純，6命令発行
レジスタ処理：128GR　128FR　レジスタリマップ　レジスタスタック
制御：バイパス，依存性検出
並列実行可能資源：4整列+4MMXユニット　2FMAC（4SSE）　2LD/STユニット　3エントリALAT
メモリサブシステム：L1，L2，L3キャッシュ
投機実行管理

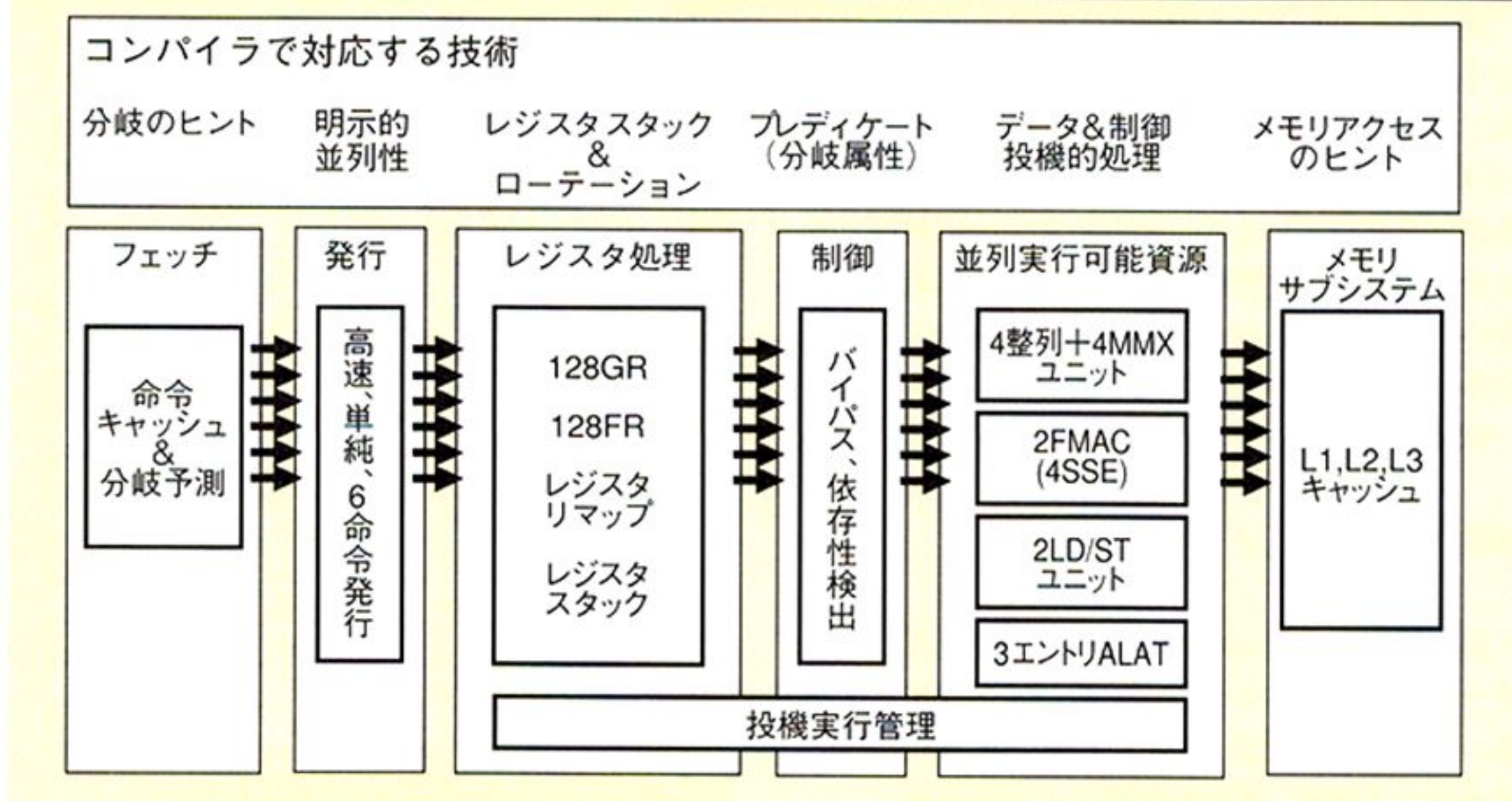



図17　Itaniumのパイプライン（10ステージ，インオーダー）

フロントエンド（前処理）　命令供給　オペランド供給　実行
IPG（INST POINTER GENERATE）　FET（FETCH）　ROT（ROTATE）　EXP（EXPAND）　REN（RENAME）　WLD（WORD-LINE DECODE）　REG（REGISTER READ）　EXE（EXECUTE）　DET（EXECUTION DETECT）　WRB（WRITE-BACK）

図18　フロントエンド

IP MUX
16KB I-Cache & ITLB
Return Stack Buffer
Decoupling Buffer
to Dispersal
Target Address Register
Adaptive 2-Level Predicator
Branch Target Address Cache
BAC1 & Loop Exit Predicator
BAC2
IPG　FET　ROT　EXP

レキュール（分子）と呼ばれ，最大4個のアトム（原子）と呼ばれるRISCに似た命令を含む64ビットまたは128ビット長の命令である（図23）。モレキュール内のアトムは並列に実行され，モレキュールの形式は直接アトムがどの機能ユニットに結びつくかを示している。これにより，デコードとディスパッチのためのハードウェアを簡略化している。

図24は128ビットのモレキュールがアトムのスロットから機能ユニットに直接対応することを示している一例である。モレキュールはインオーダーで実行され，アウトオブオーダー実行のための複雑なハードウェアは存在しない。

(3)内部構造

図25にCrusoeのCPUコア部のブロック図を示す。5つの並行実行可能な機能ユニットと内蔵キャッシュからなる単純な構造をしている。ほかのMPUがハードウェアで行っている投機実行に対応する命令列をソフトウェアで生成するため，投機実行時に参照するためのシャドウレジスタを，汎用レジスタとは別に，整数用48本，浮動小数点用16本を内蔵する。図示されていないが，Crusoeは1チップにLongRun電力管理ユニットとノースブリッジ（PC133SDRAM，DDR SDRAM，PCIバス）を集積している。

(4)パイプライン

Crusoeのパイプラインは，基本的には，Fetch0，Fetch1，Decode，Register Read，Execute，Writebackの6ステージからなるインオーダー構造をしている。一般的なx86プロセッサよりも少ないステージ数（レイテンシ）が特徴である。パイプラインは図26に示すように，各機能ユニットで若干異なる。パイプラインの最後に投機実行の終わりを明示するコミットステージが付加される場合がある。

(5)コードモーフィングソフトウェアの動作

コードモーフィングソフトウェア（CMS）は，x86命令からVLIW命令（モレキュール）へのコード変換用のソフトウェアである。これは，Crusoeが直接実行できるVLIWコードで記述されている。最初，CMSはROM（1MB程度の容量）に格納されており，MPUのブート時にDRAMに展開され，実行される。CMSが格納されるメモリ領域は，同じメモリに格納されているx86命令からは参照できない。

Crusoeからは，x86命令で記述されたOS，BIOS，アプリケーションプログラムはすべて，（変換して実行すべき）自分自身のアプリケーションプログラムに見える。そして，決められたターゲットアドレス（たとえば，x86のブートベクタである0xFFFF0番地）からコード変換を開始する。このとき，同じアドレスのx86命令を重複して変換しないように，変換後のモレキュール列を保存しておくための大容量（16Mバイト程度）のメモリ領域を使用する。これをトレースキャッシュと呼ぶ。

図27にCMSの動作の概念図を示す。CMSはまず，変換すべきx86命令の格納されたターゲットアドレスをトレースキャッシュの中から探す。もし，ターゲットアドレスがトレースキャッシュにヒットすれば，対応するモレキュール列にジャンプして，そこを実行する。もし，ターゲットアドレスがミスすれば，CMSはx86の命令列に対し，デコード，アトムへ変換，最適化，スケジューリングを行い，新たなモレキュール列としてトレース

図19　命令供給

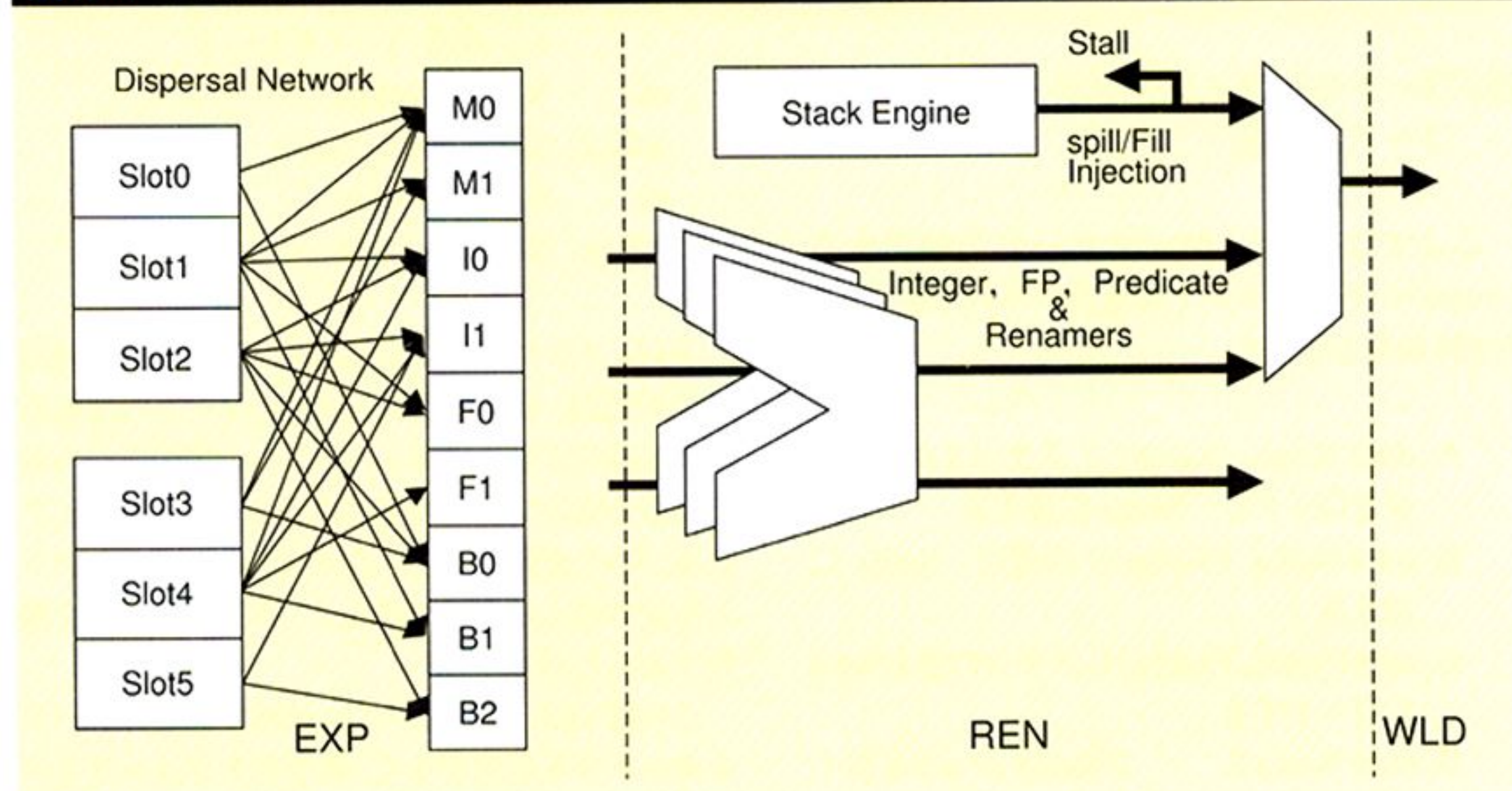



図20　オペランド供給と実行

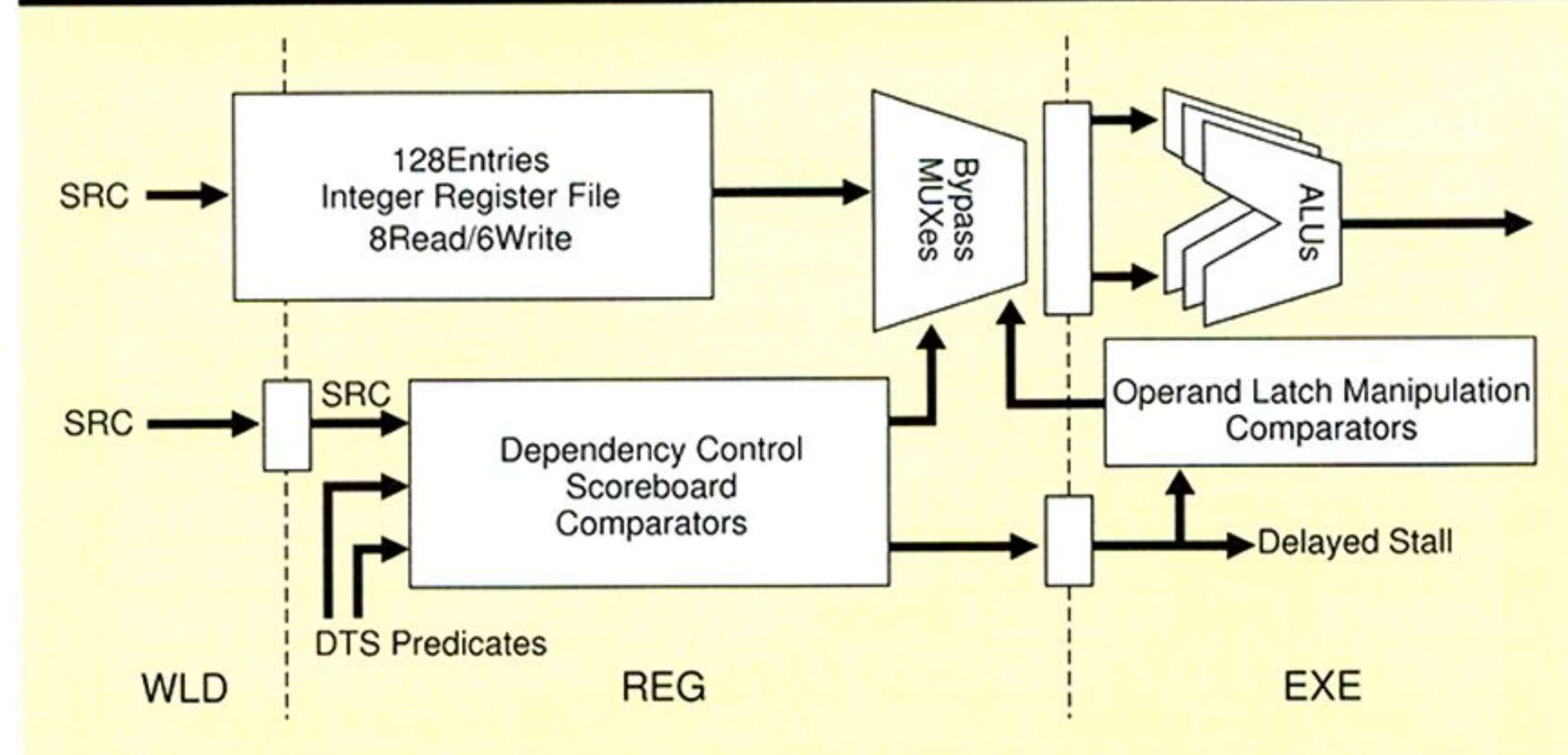
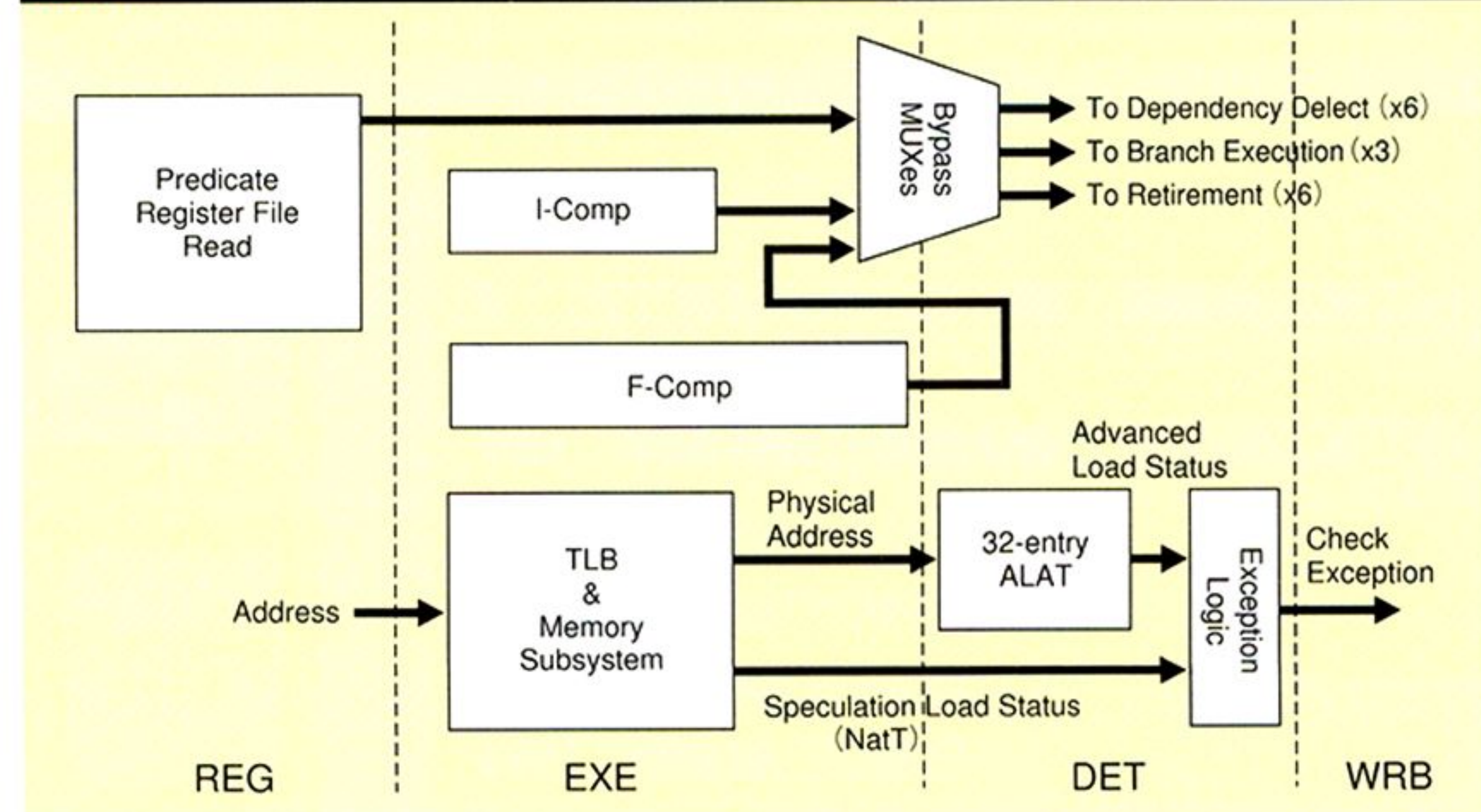



図21　例外検出

キャッシュに格納し，そこへジャンプする。

トレースキャッシュ内のモレキュール列の実行中にコード変換した最後の命令（CMSの変換処理の先頭，またはトレースキャッシュ内の他の領域へのジャンプ命令）に突き当たると，最初の処理に戻って次のターゲットアドレスがトレースキャッシュ内にあるかチェックする（直接，トレースキャッシュのほかの領域へジャンプする場合もある）。

(6) コードモーフィングソフトウェアのコード変換例

ここでは，CMSがx86コードを対応するCrusoeのVLIWコードに変換する例を示す。次のx86命令を考える。

A. addl %eax, (%esp) // スタックからデータをロードし，%eax に加える
B. addl %ebx, (%esp) // 同様に，%ebx に加える
C. movl %esi, (%ebp) // メモリの値を%esiにロードする
D. subl %ecx,5 // %ecx から5を引く

最初は，変換システムの前処理として，x86の命令をデコードしアトムの並びに変換する。レジスタ %r30 と %r31 がメモリロード操作のテンポラリレジスタとして使用され，次のように変換される。

```
ld     %r30, [%esp] // スタックからテンポラリにロード
add.c  %eax,%eax,%r30 // %eax に加算し，条件コードをセット
ld     %r31, [%esp]
add.c  %ebx,%ebx,%r31
ld     %esi, [%ebp]
sub.c  %ecx,%ecx,5
```

次は，コンパイラでよく知られている，共通部分式の削除，分岐の削除，未実行コードの削除などの最適化が行われる。ハードウェアのみからなるx86の実行では不可能であるが，ソフトウェアに基づいた変換システムなので，命令列からアトムを並べ替えるだけではなく，不要なアトムを削除することができる。

この例では，最後のアトム以外では条件コードをセットする必要がなく，命令スケジューリングの柔軟性が増す。また，ロードアトムのひとつは冗長である。

結局，アトム列は，次のように，少ない命令に落ち着く。

```
ld     %r30, [%esp] // スタックからのロードは1回のみ
add    %eax,%eax,%r30
add    %ebx,%ebx,%r30 // ロードしたデータを再利用
ld     %esi, [%ebp]
sub.c  %ecx,%ecx,5// ここの条件コードのみが有効
```

最後は，スケジューラが残ったアトムを並べ替え，レジスタ依存性がないアトムからなるモレキュールとしてグループ化する。この処理は，アウトオブオーダーなプロセッサのディスパッチ回路で行われているのと似た処理である。結局，最初の命令は次の2つのモレキュールに変換できる。

```
1. ld %r30, [%esp] ; sub.c %ecx,%ecx,5
2. ld %esi, [%ebp] ; add %eax,%eax,%r30;
   add %ebx,%ebx,%r30
```

このモレキュールはインオーダーで実行されるが，元のx86コードをアウトオブオーダーで実行するのと同じ効果があることに注目したい。また，モレキュール自体は明示的に並列性が指定されているので，単純なVLIWエンジンで実行でき，それゆえ，高速で低消費電力である。ハードウェアに命令を並べ替える複雑な機構は不要である，というのがTransmetaの主張である。

(7) 感想

Crusoeは2命令，または4命令の並列実行を想定しているので，Itaniumの6命令よりは現実的と思われる。平均的には3命令同時実行のアウトオブオーダー処理と同程度の性能（同一クロックのPentiumⅢ）と思われるが，CMSの動的な変換によるオーバーヘッドのために，その75%程度の性能に落ち着くと考えられる。これは，まずま

図22 従来のハードウェアをソフトウェアとハードウェアの組で実現

従来のx86ハードウェア	Crusoe	
	VLIWハードウェア	コードモーフィングソフトウェア
可変長命令のデコード	単純なデコード	x86のデコード
スーパースカラでの組み合わせ		命令の組み合わせ
スーパースカラでの命令発行		命令スケジューリング
バイパス回路		バイパススケジューリング
レジスタリネーム		レジスタリネーム
複雑なアドレシングモード		アドレスモード合成
アウトオブオーダー実行	インオーダー実行	
投機実行		投機実行
演算機能	演算機能	
レジスタファイル	レジスタファイル	
マイクロコードROM		ソフトウェアライブラリ
キャッシュ	キャッシュ	
FPスタック回路		FPスタック
		命令コード最適化

図23 Crousoeの命令フォーマット

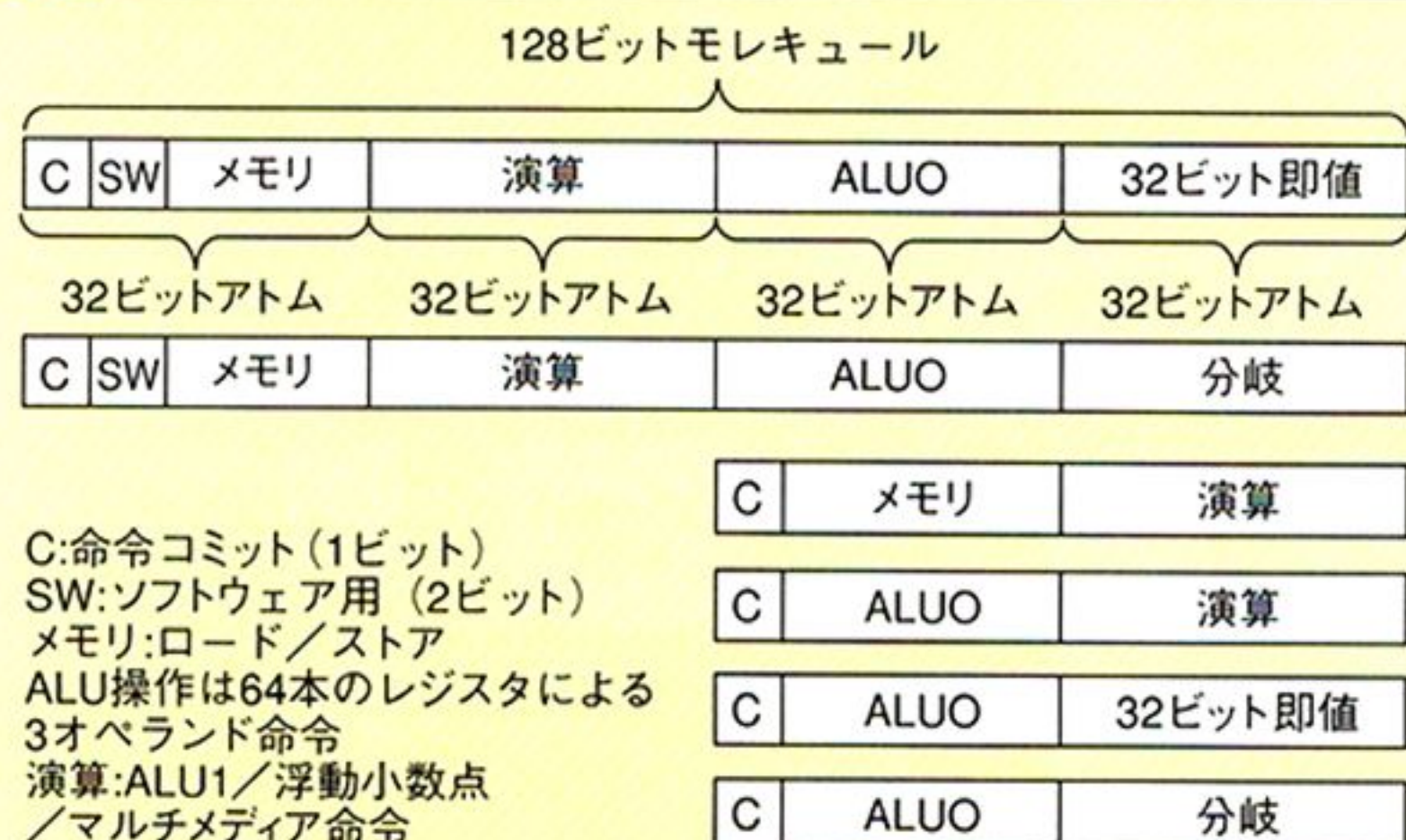
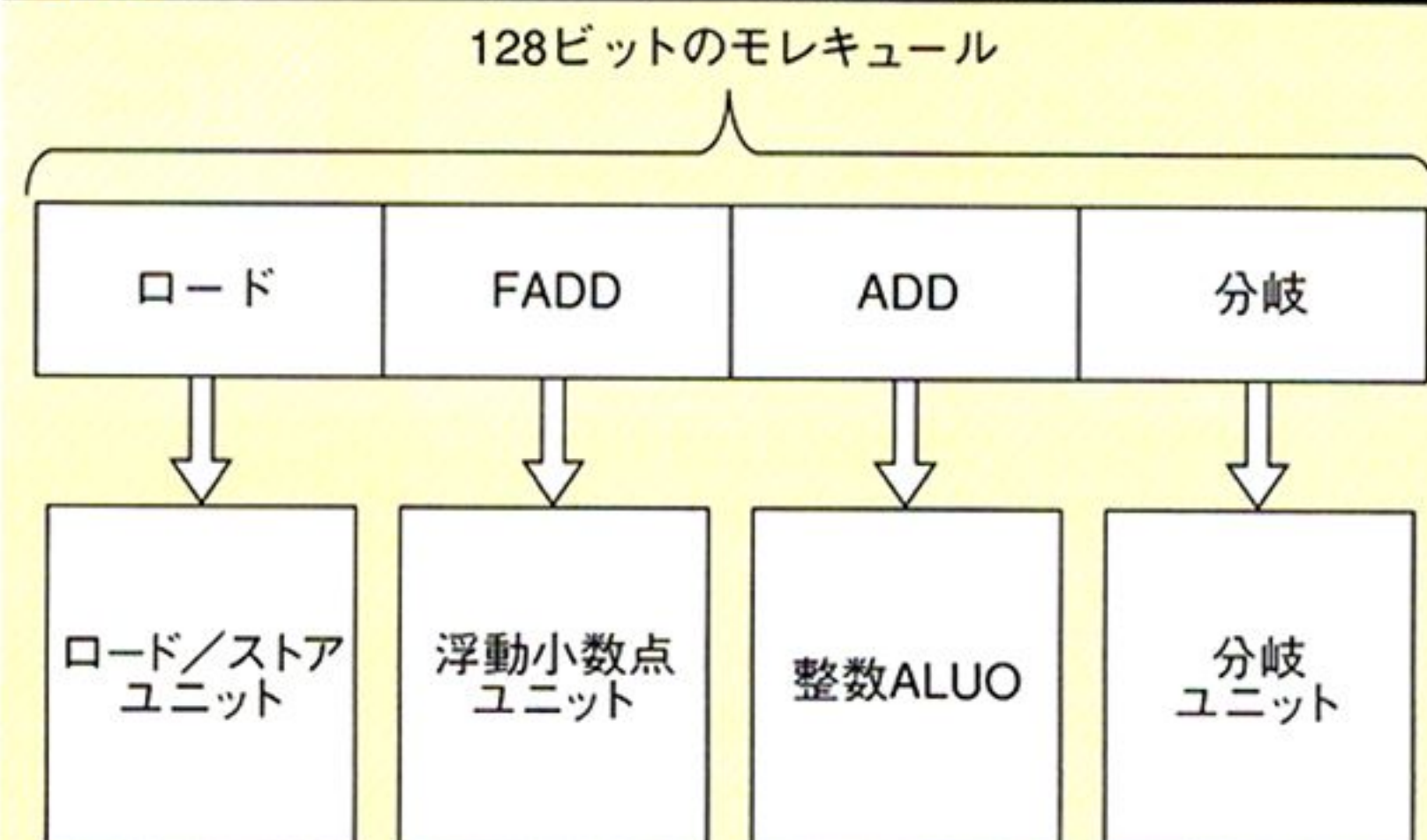



図24 モレキュール内のアトムと演算器の対応(並列に実行される)

ずの性能といえるかもしれない。エミュレーションでこのレベルの性能を達成できたのは快挙といってもよい。CMSがよほど優秀なのだろうか。それに付随するVLIWエンジンの処理性能のよさも忘れてはいけないが。

ただ，VLIWの命令形式がハードウェア資源によって規定されるため，バイナリレベルのモレキュールの互換性を保つためには，ハードウェア構成を変更できないという，VLIWマシンに共通する欠点を内在しているのは確かである。その意味で，ハードウェアによる性能改善は，動作周波数の向上とキャッシュの大容量化くらいしか道が残されていない。しかし，Crusoeの位置付けはx86命令を実行するモバイル用MPUであり，VLIW命令でプログラムを書く人はまずいないと思われる。ハードウェアの変更とともにCMSも入れ替えてしまえば，バイナリ互換性の問題は解決する。もしかしたら，Torvaldsのいうように，本当に画期的なMPUなのかもしれない。

2000年の終わりにIBMやCompaqがそのノートPCへの採用をキャンセルしてケチがついた格好のCrusoeであるが，その第一の理由は性能が「期待はずれ」だったということである（Compaqに関しては不採用は確定的ではない）。この理由はHPのItaniumの不採用にも通じるところがあるように思える。VLIWでは性能が出ないというのは定説なのだろうか。それはともかく，Transmetaは2002年にリリースするCrusoe2.0で1クロックサイクル当たりの性能を倍に引き上げ，消費電力を約半分に削減するらしい。具体的には，現在4命令128ビットのVLIWを倍の8命令256ビットに変更する。これにより，IPCが従来の2.2から5.5に向上するという。動作周波数は1GHzを越え，性能が向上したことにより動作時間が短縮され，消費電力は0.5W以下となる予定。また，同時にCMSも徐々に改版を行い性能を40％向上させるという。問題は通常のプログラムにおいて同時実行可能な8命令の組を見出せるか否かにかかっている。これはほとんど不可能なのではないだろうか。個人的には，インテルの執拗な反撃にあったTransmetaが死に物狂いでロードマップを描き直しているという気がしないでもない。

●おわりに

かつて，VLIWという方式を初めて耳にしたとき，頭に浮かんだのはマイクロプログラムというイメージだった。実際，固定長の命令，同時実行できる複数の命令をひとまとめにして処理するというのはマイクロプログラムの発想そのものである。そこで，今回はマイクロプログラミングとVLIWを同時に取り上げて対比してみたが，いかがだったであろう。違いは，マイクロプログラミングは人間の手で泥臭い方法で作成されるが，VLIWはコンパイラを使うというところであろうか。性能を出すために人間の頭脳とソフトウェア（コンパイラ）が対峙しているイメージには興味深いものがある。ただ，現在脚光を浴びているItaniumやCrusoeがこのまま生き延びていけるという意見には，個人的には，やや懐疑的であることを付け加えておく。最終的には絶対的な性能がものをいうような気がするので。低消費電力はあくまでオマケだと思う。

さて，次回は割り込みと例外について説明する予定である。それでは。

図25　Crusoeのブロック図（CPUコア）

図26　各機能ユニットのパイプライン

ALU

Fetch0	Fetch1	Decode	Register Read	Execute	Write Back	Commit
命令フェッチ前半	命令フェッチ後半	命令デコード	レジスタリード	命令実行	レジスタライトバック	コミット

ロード／ストア

Fetch0	Fetch1	Register Read	Mem0	Mem1	Write Back	Commit
命令フェッチ前半	命令フェッチ後半	レジスタリード	キャッシュアクセス前半	キャッシュアクセス後半	レジスタライトバック	コミット

FPU

Fetch0	Fetch1	Register Read	Execute	Execute	Execute	Execute	Write Back	Commit
命令フェッチ前半	命令フェッチ後半	レジスタリード	FP命令実行(A)	FP命令実行(B)	FP命令実行(C)	FP命令実行(D)	レジスタライトバック	コミット

分岐

Predict				Verify
分岐方向を予測	（待ち）	（待ち）	（待ち）	分岐方向の検証

図27　CMSの動作概念

編集室から from E・D・I・T・O・R

SHIFT BREAK

▶インターネットでパズルを検索すると，たくさんのページがヒットしますが，「コンピュータでパズルを解く」というページは少ないようです。パズルは自分で解いて楽しむもの，と思っていませんか。いえいえ，そんなことはありません。パズルはコンピュータで解いても面白いのです。今回の記事を読んで興味を持った方は，私のページ（http://www.geocities.co.jp/SiliconValley-Oakland/1680/）をご覧ください。いろいろとやっています。（M.Hiroi）

▶皆様初めまして，ねおだ如と申します。主にMacintoshを使ってイラストを描いたり，プログラムを組んだり，文章を書いたり，イベントやVPの企画を作ったりしているなんでも屋さんです。今後とも，どうかよろしくお願い致します。今回は，「この本を読んで，自力で入力して実行を確かめられるプログラム」であることに重点をおいて，MacOSのプログラム上のお約束もなにもひとまず置いて，勢いだけで組んでしまったわけですが，いかがだったでしょうか？　力及ばすなところも多いかと思いますが，お楽しみいただければ幸いです。また，Macintoshが話題のメインではありませんが，私的なサイトもありますので，興味のある方はどうぞ。また，今回の記事に関するご意見・ご感想などもお待ちしております～。（http://www.sagami.ne.jp/~ltjg/）（如）

▶昨年は，Palm機を買い，WonderWitchを買い，さまざまな開発環境で遊ぶ機会が増えました。Palmなんて，gccで開発できるんですね。そうして見渡してみると，世の中には良質のフリーの開発環境がたくさんあります。PerlやRubyはいうに及ばず，それ以外にもいろいろな開発環境が公開されています。いろいろ触ってみると，それぞれの特徴が見えてきて面白いです。

さて，FutureBASIC3日本語版が昨年１２月に無事発売され，いよいよ日本でもFB3製のソフトが現れつつあります。これからの盛り上がりに期待したいです。（水野貴明）

▶さんざん期待させてこの内容。反省しております（DirectX8に手を出したのが敗因ともいう……）。続きはインターネットからのコンテンツで補完させていただきます（サポートページ http://galaxyknights.com/ 作りました）。一緒に手伝ってくれたねおだ氏や，編集の（U）さんにも，ご心配をおかけしました。この場を借りてお詫びします。（よ）

▶昨年末，UnderWorldというテクノユニットが出した“EVERYTHING，EVERYTHING”というDVDライブアルバムを観たいがためにDV環境を構築しました。48トラックのデジタルレコーダを２台駆使して録音された音源，そしてアートを担当しているTOMATOのクールな映像が絶妙にリンクされていて，凄いのひと言です。現時点では，音楽・映画そしてゲームを含めても，これだけDVDの特性を生かしきったものはないと断言できます。特に映像関係に興味がある方は要チェックです。で，話しは変わりますが，今回のOh!Xのお仕事が終わったら，東京へ引っ越そうと思っていたのですが，住む場所がなかなか決まりません。割と都心へのアクセスがよくて，家賃がそこそこのところを探しているのですが難しいところですね。それに根が田舎者の私には，東京は少し怖いところだったりして……。（飯田）

▶今年のお正月，一通の手紙が届きました。それは15年ほど昔，21世紀の自分に宛てた手紙でした。これは，昭和60年に開催された国際科学技術博覧会を記念して行われたもので，当時小学生だった私は，そのとき夢中だったアイドルのことを書き，未来の私に手紙を書いたのです。とても恥ずかしい内容で赤面してしまうのですが，こんな頃もあったのだとちょっとうれしくなる21世紀初めの出来事でした。

今回のエッセイでは，お芝居をテーマにしています。家の近くに劇場があるので，ここ数年よくその劇場に足を運んでいます。

昨年のクリスマスには，シェイクスピア原作の「リチャード三世」を観てきました。リチャード三世には俳優の大地康雄さんが演じておられ，とても迫力のある演技で，小説などのイメージとは少し違ったリチャード三世を見ることができました。

ところで，今回からちょっと趣を変えてShadeのプラグイン開発に関する記事に挑戦してみました。技術的なことや数学的なことはあまり得意ではないので，ちょっと，わかり難いところや，記述が正確でないところがあると思います。（U）さんから，続きを書いてもよさそうな返事をいただいたので，今後はもっと質の高い記事を目指してがんばりますので温かい目で見守ってください。（田中）

▶前の原稿から今回の原稿の間に，会社を辞めて，バリ島に行って，結婚して，ディズニーランドに行って，子供を産んでしまいました。次回は学生に戻る予定です。
（絵美　http://www.antai.suginami.tokyo.jp/emi/）

▶本誌の発売が大幅に遅れたおかげで，DOGA-L1，L2，L3はバグを修正したバージョンを収録することができました。でも，BGMジェネレータの記事は，新製品紹介だったのに，もう全然新製品ではなくなってしまいましたね。

●13thCGAコンテストの応募締め切りは，２月14日(水)です。今年もすばらしい作品を多数応募ください。なお，東京上映会は４月29日（土）なかのZERO大ホールです。

●今年からちょっと面白いプロジェクトをスタートします。３月の某ショーで派手に発表する計画。ご期待ください。次号では，そのプロジェクトの裏舞台でも記事にしようかなぁ。（かまた）

▶うーむ。予約してしまいましたよ，NTTドコモF503i。そう，Java対応iモード端末というやつです。来週にはいよいよ，ケータイ電話でプログラミングができる日がくるのでありますねぇ。ほかにもJ-PHONEも別系統のJava端末を出すし，５月にはFOMA（IMT-2000）もありますし，今年はケータイイヤーであります。いや，というか，去年もケータイイヤーであり，常時接続イヤーであり，MP3イヤーだったわけですけど。

趣味のコンピュータ方面の楽しみ方って，ここ数年，世間では自作PCを組み上げーの，ビデオチップがこう，クロックアップがどうという方面のみがクローズアップされる傾向にありましたが。なんか，こうなるといろいろ多様な楽しみ方が市民権を得ることができそうで楽しみです。うんうん，多様化ってのはいいことですな。私が今まで日の当たらなかったケータイマニアの常時接続野郎，ID付きスマートメディア愛好者であり，Spedia野郎だからというわけではないですが。さぁ，今年の流行はレンタルサーバーを借りてiモードJavaゲームサーバの運営だっ（嘘）。

それはそれとして……，今回は，iモードJava端末の発売と重なったりしたために編集さんには

多大な迷惑をかけてしまうはめになってしまいました。すいません。最近，私はなぜかケータイ系ライターってことになってるせいですが，ひと月でこんなにたくさんの原稿を原稿を書いたのは初めてです。すごいよなぁ，iモードブーム。……でもでも，俺も精一杯がんばってるんで許してくださいっ。だって，お正月だって休んだのは元旦午前0時から2日の箱根駅伝第2区までだったんですよぉっ(いや，それは，単に母校が駅伝でリタイヤしたからだ)。（で）

▶20世紀中に出るはずだったのに〜，というのは誰しも書くことだろうから省略。今号では自分がなにを書いたのかすら忘れてしまうほど。私が謝るべきことかどうかはわからないが，作っている立場の人間としてひと言。すまん。そういや遥か昔に，2000年のクリスマスにパッケージゲームを作るとかもいっていた記憶があるが，予想通りそちらも頓挫。そうこうしているうちに，とある会社でとある仕事を始めてしまい，今年は時間もあまり自由が利かなくなりそう。原始的と思いつつも，とあるスクリプトを黙々と書き倒さなければならなかったり，一緒に働いている娘に年甲斐もなく恋煩いしてなにも手につかなかったり，なにせ2人だけのプロジェクトだけに雰囲気が悪くなると仕事ができなくなるといいわけしてみたり。新世紀早々，前途は多難らしい。別にいままでが悠々ってわけじゃないけどね。そんなわけでIMは今回もお休み，RMは飯田君に譲った（押しつけたともいう）が，彼はちゃんとやってくれたろうか？（i.k）

▶結局，前号の「PS2の研究」は序章だけだったか。ということを考えると序章だけで終わった「真・仮面ライダー」を思い出す。怪奇路線に回帰して大人向けのストーリー展開を目指した割にはイマイチなできだった。で，いまは「仮面ライダークウガ」が面白い。最初見たときはストロンガーのような不細工なデザインだなと思ったが，慣れてくるとなかなか味がある。ストーリーも丁寧に作ってあるところがいい（武器マニアにはディテールの甘さが目につくようであるが）。最初のうちは話の展開が遅いのと残酷な場面が多い点で打ち切りの心配もしたが杞憂だったようだ。敵の親玉が同じ形態という設定はどこかで見たような気もするが，いまお気に入りのTV番組のひとつだ。本号が発売される頃には最終回を迎えているはず。五代は戦うだけの生物兵器になってしまったのかな？

ところで，小波のやることは理解できない。「デジタルビッグコミックスピリッツ」や「パトレイバー」などを商標登録すると，版権にうるさい小学館と喧嘩になるのは必至なのに。もしかして，劇空間プロ野球の騒動で味を占めたか。個人的には「ホワイトベース」と「木馬」の登録に腹が立つ。こちらはバンダイが怒りそうだが。関係ないが，「イデオン」で「メイフラワー」が「若草のシャルロット」の商標に抵触するということで「ソロシップ」に名称変更された話を思い出してしまった。ララァの「エルメス」の名称もプラモでは使用できなかったんだっけ。ファミ通をボイコットする話を聞くと思い上がりもはなはだしい。ファミ通側も関係修復の努力などせず，放っておけば拍手ものだったのに。結局，和解しちゃったみたいだね。ああ，「パロディウスだっ！」の頃はよかった。いまじゃ，「大波」，「小波」はコナミの商標です，ってか。なお，プロ野球選手の肖像権の独占については，日本プロ野球選手会から「独占契約は認めていない。無効だ」との声明が出たと11月23日のasahi.comで報道された。当然だね。

著作権関連でもうひとつ。正月番組を観ていたら水木杏子といがらしゆみこの係争はまだ続いていて「キャンディ，キャンディ」の画像をテレビ放映できないらしい。決着はいつなんだろう。

（幸恵ちゃん，ザギグビザジョ，のA.N.）

▶まだまだ佳境はこれからなのだけど，とりあえず編集後記だ。

とりあえず発売が遅れに遅れたことについてお詫びしたい。少々遅れるのはいつものことかもしれないが，ちょっと間が空きすぎた。記事によってはほぼ1年前に書かれたものもある。まとまった空き時間がとりにくくなって，いろいろタイミングを調整していくのが難しくなった。もうちょっとうまく調整しないといけないだろう。

グランツーリスモ3がまた延びたらしい。全然人のことはいえないのだけど，遅れの幅にするとOh!X以上なのかあ。それが許される体制というのはほかと比べて相当恵まれているんだろうなあ。（U）

春夏秋冬 ひと巡り

当初はあまり間を置かず，初夏くらいに……と思っていたのですが（甘い甘い）。夏がきて秋がきて，なんとかちょっと遅らせれば2000秋号でいけそう……というところで，また延期になりまして，2000冬号，2001冬号と変遷しつつ，最終的には2000春号の次が2001春号になってしまいました。季節はまだ冬なのですが，冬から始めるのもいかがなものだろうかということもあり，初春ということで春号とさせていただきました。

今年度は雑誌の仕事からはなれていたのですが，とりあえず二足のわらじ状態はどちらにとってもよい影響がないことがよくわかりました（反省）。次回はそれを踏まえてもうちょっと効率よくやれる体制を考えていきたいと思っています。

ざっと読んでいただければわかるとおり，今回のOh!Xでは実にたくさんのものが抜けています。STUDIO Xがなく，知能機械概論がなく，ゲームレビューがありません。そのほかの連載もいくつか抜けていますし（かと思うと，束になっている連載記事もありますが），細々したものはまったくありません。体裁としては完全形ではないのですが，ばたばたしているあいだにとりあえずこんな感じでまとまりました。

もっと薄い本でもいいのではないのかという意見もあるでしょうが，薄くしたからといってさほど楽になるわけでもありませんし，結局いろいろやっていくと要素は足りなくてもこれくらいにはなってしまうことになります。次回はもう少しバランスを重視したいところです（再び反省）。

●正式ページオープン

すでにご存じの方はご存じでしょうが，Oh!XのWeb正式Webページがオープンしました。以前あったページは正式ではないのかといいますと，あまり正式なものではありませんでした。半ゲリラ的にサーバ内にディレクトリ掘って置いておいたページでしたので（普通はこういうことすると怒られると思う），あまりおおっぴらにはできませんでしたが（してた気は少ししますが），今回初めてソフトバンクパブリッシングのほかのページからのリンクでもたどり着けるようになりました。

アドレスは，
http://www.vwalker.com/publishing/OhX/webx/
となります。ディレクトリの大文字小文字にご注意ください。

まだコンテンツはほとんどないのですが，Oh!X関連の各種情報はここから発信されることになります。特に次号の発売日などの情報，そしてCD-ROMに載せきれなかったファイル群のダウンロードやファイルアップデートなどの窓口になりますので，ときどきチェックするようにしてください。

また，同ページを構築していくためのボランティアスタッフも募集しています。興味のある方はWeb上での告知をよくお読みのうえメールまたは掲示板でご連絡ください。

microOdyssey

20世紀のうちに出しておきたかった本をようやく出すことができそうだ。

ばたばたとしているうちにすっかり時は21世紀。無論，キリスト教徒でもないので，ミレニアムも21世紀も私には関係ないのだが，それでも小さな頃から「21世紀」というのはどういうものかというのを叩き込まれてきているのも確かなのだ。そう，それは恥ずかしいくらい「輝く未来」。

実は21世紀などというものは，漫画にせよSFにせよ，結果的に陰鬱で否定的な描き方をされていることのほうが多いのではないかと思うのだが，インプリントされているのは不思議と明るい未来だけだ。個人的にいうと私の21世紀のイメージは1970年万博にかなり近い。詳しくは覚えてなくても，日本も元気のあった時期だと理解している。いま思えばバタくさい面もあるが，Wonderの宝庫でもあったのだ。あんなにいいイベントというのはほかにお目にかかったことがない。おそらくよいことだけではなかったのだろうが，時とともに浄化されてよいイメージだけが残される。それはおそらく新世紀が明るいものであってほしいと願っているから。

そしていま「光り輝く未来」と呼ばれたものを目の前にしている。なにはともあれ21世紀がやってきてしまったのだ。

生まれたときから世紀末なのはしかたないにしても，初めて迎える新世紀。歴史の繰り返しを見る限り，不思議なもので世紀末だとちゃんと退廃が進み，末法の世だとなれば本当に世が乱れる。新世紀にはちゃんと新世紀らしい革新が起こった事例も多い。いわば世界レベルのプラシーボ効果だろう。論理性はともかく，人の思い込みの力は立派に歴史を大きく動かしていく。

いまさらいうまでもないことだが，カレンダーが21世紀になったからといってなにが変わるというわけではない。神はいないと思うが，7時ともなれば揚雲雀名乗りいでカタツムリは枝を這う。だが，単に時間の区切りだけではなく，そこに別の意味をつけられるのが人間というものだ。21世紀だから21世紀らしいことを，と誰もが望むことで本当の21世紀はやってくる。自分が作りたい21世紀は誰に与えてもらうものでもない。

ホンダのASIMOやソニーのAIBOなどは「おい，もう21世紀だぞ。そろそろロボット作らなきゃ」といった技術者の気概が感じられて好感が持てる。シャープはちゃんと普及型壁掛けテレビを作って元旦から売り出した。通信回線も20世紀とは桁違いに速くなりつつある。それはよい傾向だ。

しかし，コンピュータはまだしゃべらない(少なくともちゃんとは)。人工知能，自然言語解析，スピーチシンセサイザや音声認識は少しずつ進歩しているのだろうが，根本的になにか違うと感ずるものがあるのは，その延長上にあるべきものが，なかなか見いだせないからかもしれない。具体的なイメージの実現は，ときとして開発者のエネルギーを驚異的に引き出すことができるようだから。

パソコン(というよりは単にコンピュータか)が21世紀にはいかにあるべきか，ゲーム機がいかにあるべきか，など，意外と誰もに共通したイメージは擦り込まれているのではないかと思う。

ロボットについては誰もが鉄腕アトムを目指しているという説もあるが，それだけでもないだろう。ムーバブルフレームはどう作るべきかとか，50mを超えるものをいかにして合体させる必然性を導くかなど研究テーマにはこと欠かない。我々はたくさんの未来をすでに託されているのだ。それらは誰かが始めなければ絶対に実現はしない。誰かがやるんじゃなくて自分からやらなくちゃ始まらない。その流れがあってこそ新世紀が開かれるというものだろう。やがて人間と対等にしゃべり始めたコンピュータが「～ですぅ」と舌足らずだったとしても，誰を責めるわけにもいかない。

ここでもうひとつ問題を提議するとしよう。では，そのロボットなり壁掛けテレビは，本当に自分たちの望んだものなのか？　理屈(？)では，トヨタあたりはエアカーを作らないといけないのではないかと思う。車輪もなく翼もなく自動車形状のまま空を飛ぶものはちょっと難しいにしても，ホバーで浮くくらいはできるだろ？　という気はしなくはない。高速運行すると凄く危険なのは目に見えており，燃費も悪そうだ。道がなくても走れそうかもしれないが，段差があるとうまく走れそうにはない。登坂能力もあまりなさそうだ。それよりはエコロジーなハイブリッド自動車を作るというのは，他人の未来ではなくて自分の未来を選んだ選択といえるだろう(でもエアカーの試作くらいはすべきだよな)。

託された未来がたくさんあれば，どれかを選ぶ必要もある。「パソコンに向かってしゃべりかけるなんて絶対ヤだよね」「本当に脳に電極埋め込みたいの？」など，個人差は激しいので取捨選択されるのが当然。自分で選んで，あとはそれぞれ頑張ればいい。信じている限りはそれが自分の未来だ。

はっきりいって，与えられた21世紀にはあまり興味がない。これから切り開いていく21.5世紀があくまでもターゲットである。かねてから述べているとおり，Oh!X自体の進む道はすでにそこに向けて設定されている。それが我々にとって本当の21世紀だと思うから。

さて，ここで改めて問おう。21世紀のコンピュータはこんなままでいいのか？　(U)

次号予告

Oh!X 2001 秋号(?)　2001年秋発売予定

特集　未定(3D関係か？)

(以後の号はまったく未定です)

発行予定など詳しい情報は，

http://www.vwalker.com/publishing/OhX/webx/

で確認するようにしてください。

Oh!X 2001春号
/Vmag.Online編集部　Oh!X制作実行委員会編

■2001年3月1日発行　定価2800円(2667円+税)
■発行人　稲葉俊夫
■編集人　来島　樹
■発行所　ソフトバンク パブリッシング株式会社
東京都港区赤坂4-13-13　東亜赤坂ビル
販売局　☎03(5549)1201
■デザイン　クニメディア(株)
■印刷　クニメディア(株)

乱丁，落丁，CD-ROMの破損はお取り替えします。小社販売局までお問い合わせください。
雑誌　65815-51

Oh!X 2001 春号

愛読者カード

郵便はがき

料金受取人払

赤坂局承認

604

差出有効期間
平成14年4月
30日まで

1078790

(受取人)
東京都港区赤坂
4丁目13番13号

ソフトバンクパブリッシング(株)
Vmag Online 編集部 内
Oh!X 2001 春号 係行

フリガナ	年齢	性別
お名前	歳	男・女
ご住所 〒		
e-mailアドレス： TEL URL：		
勤務先名 使用機種/OS (学校名)	パソコン 使用歴 年	
購読パソコン誌		

2001春号通巻172号

電脳倶楽部CD版やパーフェクトコレクションなど，在庫販売しております。

満開の電子ちゃん

作・え 岡村祭

電子ちゃんの「出た」という台詞がありますが、鉄の大陸は、まだ開発中です。「今春」にはなんとか…。

二〇〇一年某月
宇宙人来襲

今週来ても「来襲」なんちゃって
おお!
紳士的な宇宙人だ
よかったよかった

家畜 農作物 水 石油 ウラン 全部いただきます
地球人は全員奴隷 逆らう人は殺します
うわー!
紳士的なのは言葉づかいだけだ

なまける人は殴ります
働け働け
弱点はないの? やつらの
ここだけの話だけど

X68000? なんだそれ
なんでも昔出てたパソコンで
あっ聞いたことある 偶然それにさわった宇宙人が溶けちゃったって
それだ

作れ作れ
ねじ回し貸してくれ
チップが足りんぞ
この基板黄ばんでる

こうして各家庭に一台ずつのX68000が行き渡った
やつらが逃げて行くぞ
宇宙にはまだまだ解明されぬ謎が(以下略)
結局どういうわけだったんだ

しかしこの残ったパソコン 何に使えばいいんだ
いいソフトがあるわ! 満開製作所から出たRPGで「鉄の大陸」!

人と森の物語がテーマで敵との戦闘もアツイし、データはフリー素材として使えるの 満開の思い出にぜひ一枚どうかしら!
この68 全部CD-ROMドライブついてないぞ
急いでたからねえ

Muff&Huff
水野裕之音楽作品集
3,500円(税込)
好評発売中

Muff&Huff
MIZUNO Hiroyuki

2月発売予定:7,800円(税込)
X680x0用RPG

鉄の大陸
IRON CONTINENT

このたび満開製作所はX68関連事業を終了することとなりました。
なお、現在開発中の鉄の大陸の発売、また既発売の製品在庫販売は当面の間継続していく予定です。在庫についてはお問い合わせ下さい。
長らくの御愛顧、まことにありがとうございました。厚く御礼申し上げます。

東京都豊島区池袋2-24-2 メゾン旭1304
03-3985-6110／FAX03-3985-5366
詳細は、http://www.mankai.co.jp/で御確認下さい。

9784797315356

1929455026672

Printed in Japan　クニメディア

ISBN4-7973-1535-0　C9455 ¥2667E

雑誌 65815-51

UNIX 2001春号　ソフトバンク パブリッシング株式会社

定価 本体2,667円 +税